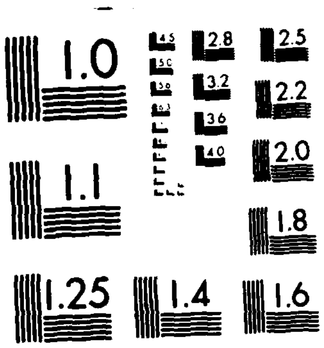
AD-A126 979 ANALYSIS OF THE RELATIONAL DATA BASE MODEL IN SUPPORT OF AN INTEGRATED APPLICATION SOFTWARE SYSTEM(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA R NISHIMURA DEC 82 1/2

UNCLASSIFIED F/G 9/2 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

2

AD A126979

NAVAL POSTGRADUATE SCHOOL

Monterey, California

DTIC
ELECTE
APR 19 1983
D
H

THESIS

ANALYSIS OF THE RELATIONAL DATA BASE MODEL
IN SUPPORT OF AN
INTEGRATED APPLICATION SOFTWARE SYSTEM

by

Rodney Nishimura

December, 1982

Thesis Advisor: Dushan Badal

Approved for Public Release; Distribution Unlimited

DTIC FILE COPY

83 04 18 002

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO. AD-A126 979

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
Analysis of the Relational Data Base Model in Support of an Integrated Applciation Software System.

5. TYPE OF REPORT & PERIOD COVERED
Master's Thesis
December, 1982

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
Rodney Nishimura

8. CONTRACT OR GRANT NUMBER(s)

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Naval Postgraduate School
Monterey, California 93940

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Naval Postgraduate School
Monterey, California 93940

12. REPORT DATE
December, 1982

13. NUMBER OF PAGES
130

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

15. SECURITY CLASS. (of this report)
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Approved for Public Release; Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Relational Data Base Model, Integrated Application Software

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
The premise of this thesis is that many software application systems perform similar functions on a data object and contain a significant operational intersection. An Integrated Application Software System (IASS) integrates the capabilities of the applications into one system. The purpose of this thesis is to evaluate the utility of the relational database model to conceptually integrate the text processing, relational database management, form
(Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT (Continued) Block # 20

generating, electronic mail, and electronic modeling applications.

The conclusion of this study is that the relational database model can conceptually support the data representation and manipulation requirements of each application considered. Furthermore, the integrated system has potential capabilities that are not available in the non-integrated set of applications.

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Avail and/or
Dist Special
A

DTIC
COPY
INSPECTED
2

Approved for public release, distribution unlimited.

Analysis of the Relational Data Base Model in Support of an Integrated Application Software System

by

Rodney Nishimura
Lieutenant, United States Navy
B.S., University of Southern California, 1975

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1982

AUTHOR: Rodney Nishimura

APPROVED BY: ______________________________
Thesis Advisor

____
Co-Advisor/Second Reader

____
Chairman, Department of Computer Science

W M Woods
____
Dean of Information and Policy Science

ABSTRACT

The premise of this thesis is that many software application systems perform similar functions on a data object and contain a significant operational intersection. An Integrated Application Software System (IASS) integrates the capabilities of the applications into one system. The purpose of this thesis is to evaluate the utility of the relational database model to conceptually integrate the text processing, relational database management, form generating, electronic mail, and electronic modeling applications.

The conclusion of this study is that the relational database model can conceptually support the data representation and manipulation requirements of each application considered. Furthermore, the integrated system has potential capabilities that are not available in the non-integrated set of applications.

TABLE OF CONTENTS

I. INTRODUCTION 8

A. APPLICATION SOFTWARE PROBLEM 8

B. THE IASS OBJECTIVES 9

C. IASS APPLICATIONS 11

II. IASS DATA OBJECT 13

A. THE TABLE 13

B. IASS TABLES 14

1. Text Processor Tables 18

2. Database Tables 20

3. Form Generator Tables 22

4. Electronic Mail Tables 25

5. Electronic Spread Sheet Tables 29

III. CONCEPTUAL INTEGRATION 33

A. OVERVIEW 33

B. BASIC IASS PRIMITIVES 34

1. Insert 35

2. Modify 35

3. Delete 35

4. Project 36

5. Select 36

6. Union 36

C. REALIZATION OF LOGICAL OPERATIONS 37

1. Text Processor/Form Generator 37

2. Electronic Mail 40

3. Electronic Spread Sheet 44
IV. IASS EXTENSIBILITY 53
A. COMBINING IASS TABLES 53
B. INTRA TYPE COMBINATIONS 54
1. Text/Form 54
2. Mail 57
3. Spread Sheet 58
C. INTER TYPE COMBINATIONS 60
1. Text 60
2. Form 64
3. Mail 66
4. Spread Sheet 68
V. CONCLUSION 71
A. FINDINGS 71
B. FOLLOW-ON RESEARCH 72
LIST OF REFERENCES 75
APPENDIX A (Word Star) 76
APPENDIX B (Vi) 85
APPENDIX C (Edit) 92
APPENDIX D (Nroff -Me) 96
APPENDIX E (Dbase II) 100
APPENDIX F (Sequitur) 106
APPENDIX G (Visicalc) 114
APPENDIX H (Zip) 121
APPENDIX I (Mail) 124

BIBLIOGRAPHY 129

INITIAL DISTRIBUTION LIST 130

I. INTRODUCTION

A. APPLICATION SOFTWARE PROBLEM

The utilization of computers in many areas, such as personal computing or office and manufacturing automation, is rapidly expanding. No longer is their use being relegated to support personnel, but is spreading into the ranks of lower and middle level management. The majority of such users are non-computer professionals who are coming to depend on the computer to provide support to accomplish their primary responsibilities.

Over the past years, numerous software packages have been made available to support a broad spectrum of users in varying environments. Capabilities such as word processing, database management, modeling, form generation, and electronic mail have become essential. The purpose of introducing the computer into an organization is to increase effectiveness and efficiency. While the performance of each support package is individually satisfactory, the manner in which they are presented to the user as a group is not. As illustrated in Figure 1.1, each support system is typically disjoint from all others, and the user is presented with different models, command vocabularies, and operating instructions. This non-integrated combination of

application software requires a special effort on the part of the user to learn a new system and remember it along with the other systems that are used. For instance, one simple task can be invoked in a different way in each system.

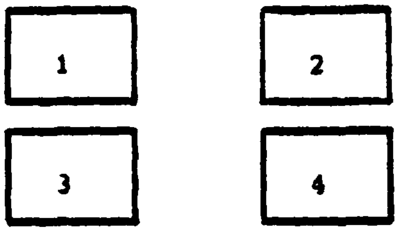


Figure 1.1 - Disjoint Support Systems.

B. THE IASS OBJECTIVES

What is needed to increase productivity is an integrated system that combines the capabilities of the applications and presents the user with a single, yet easy, conceptual data model and vocabulary set. It is such a system that is called an Integrated Application Software System (IASS). The objectives of such a system are:

(1) Ensure a high degree of user friendliness and emphasize simplicity.

(2) Minimize the initial and acquired user skill level necessary to use the system.

(3) Minimize the learning time required to use the system.

(4) Present a logical distinction between each of the IASS's capabilities, but minimize explicit navigation between them.

(5) Realize the largest functional intersection of the capabilities of each included application.

(6) Develop a minimum set of primitive commands.

(7) Minimize the dependence on programming in order to use the system.

(8) Embody the notion of software adaptivity whereby the user can learn a new application by learning only a small increment of new application specific commands and functions.

(9) Embody the notion of software reusability. New applications can be implemented by adding a small increment of functions and commands which can be expressed in terms of existing IASS operations.

While the IASS cannot be expected to completely integrate the features of each support package, it can strive to maximize the intersection between them. Figure 1.2 shows a simple illustration of an IASS in a Venn Diagram.

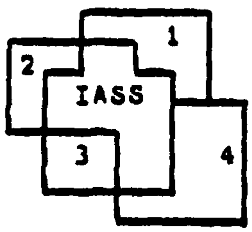


Figure 1.2 - IASS Intersection.

The purpose of this thesis is to determine the utility of the relational database model as the kernel of the IASS.

C. IASS APPLICATIONS

Five common software applications were selected to be integrated:

(1) Text Processor

(2) Relational Database Management System

(3) Electronic Spread Sheet

(4) Forms Generator

(5) Electronic Mail

As a non-integrated collection of application software, each is implemented to accomplish a predefined set of operations on a specific file type. Data in a file is not directly sharable between applications and neither are the commands to manipulate the data. Command vocabularies are usually "baroque" in that most of the operators are intended to exist as a matter of convenience to the user. However, too often it is a very small percentage of the overall vocabulary that is used most of the time. Users usually learn a subset of the vocabulary necessary to accomplish the essential functions of the application, and disregard the rest. It is the intersection of functions and vocabularies that the IASS subsumes and makes the common system for all included applications.

Commercially available software application packages were reviewed to determine the nature of the logical file types and the essential functions. A detailed description of each of the application packages is included in Appendices A through I.

II. IASS DATA OBJECT

A. THE TABLE

The logical file of any application system contains data which is used by a specific set of application programs. The key to achieving an Integrated Application Software System (IASS) which can support each logical file type is to map each into one data object. The data object chosen for the IASS is the table since it is a natural method of organizing data and is an easily understood object. Each column in the table represents one attribute of the file and each row represents an unique occurrence. The tables include columns which represent key values to uniquely identify each row. Any datum in a table can be accessed by specifying the name of the table, the value of the key, and the name of the attribute containing the datum. In this thesis, rows will will also be referred to as tuples or lines and columns as attributes or fields. A complete description of the table is given in Martin [Ref. 1].

This chapter describes the preliminary design of a set of tables which can support the data requirements of the IASS applications.

B. IASS TABLES

In the IASS each application is a logical database consisting of a set of tables. There are three general classes of tables, data table, application directory, and data table schema. The data table represents the logical file of an application. The data tables are typed according to their primary use as, text, form text, database, spread sheet, and mail. Data table typing is done only to logically organize data tables which are used primarily by the same application programs. The IASS does not support strong data table typing. Being able to combine tables of different types is an important feature of the IASS.

The application directory table contains descriptive and definitional data about the data tables in an application or logical database. Each row in the application directory table describes one data table and has a standard schema, Figure 2.1. ID is the primary key value of the application directory table. NAME is the unique name of the data table. COLUMNS is a list of column names in the data table. This list implicitly defines the schema of the data table. ACCESS CONTROL defines the access privileges of various classes of system users to the table, including read and write access and privileges to modify the data table schema. This data item also contains information concerning which operations are allowed on the data table. For instance, it may be decided that one database data table cannot be joined

with a mail data table. TABLE POINTER points to the data table. DESCRIPTION is a literal description of the table. VIRTUAL indicates whether the data table is composed from other IASS data tables. CONDITION indicates how a virtual table is composed. For example, if the virtual data table was formed by a join, that data would be stored in the CONDITION column of the application directory. GLOBAL contains all the recurring information which is applicable to a data table as a whole, such as formatting, display mode (e.g. page or table), access paths, etc. For each application, the schema of the application directory table can be augmented as required.

ID	NAME	COLUMNS	ACCESS CONTROL	TABLE POINTER	VIRTUAL	CONDITION	GLOBAL	DESCRIPTION

Figure 2.1 - Application Directory Table Schema

The data table schema table contains a row for each column in the data tables. The schema of the data table schema table Figure 2.2, is the same in each application. ID is the primary key of the table. Each column in the a data table has an unique NAME. TYPE and WIDTH describe the

data type associated with the column and the maximum width of the data entry.

ID	NAME	TYPE	WIDTH	SYNONYM	TABLE	ACCESS CONTROL

Figure 2.2 - Data Table Schema Table

Data typing supports the data integrity function of the underlying system. SYNONYM is the list of names by which the column could be referred. This information is used to determine relationships that exist with other data tables, possibly of a different type or allows the same column to be referred to by many different names depending on the context of its use. For example, in a personnel data table, the column name may be PNAME. This column could be referred to as PERSNAME, NAME, PERSON, etc. This column must be used with caution. Its data may inadvertently change the access privileges of a user to a data item. TABLE is the data tables the column occurs in. A query of the data table schema table can be done to determine the names of the tables the column is in to avoid searching through an entire application directory. ACCESS CONTROL defines the privileges associated with each column for a class of system

users including read and write access and privileges to modify the column definition.

Figure 2.3 shows the relationships between the tables that exist in each application.

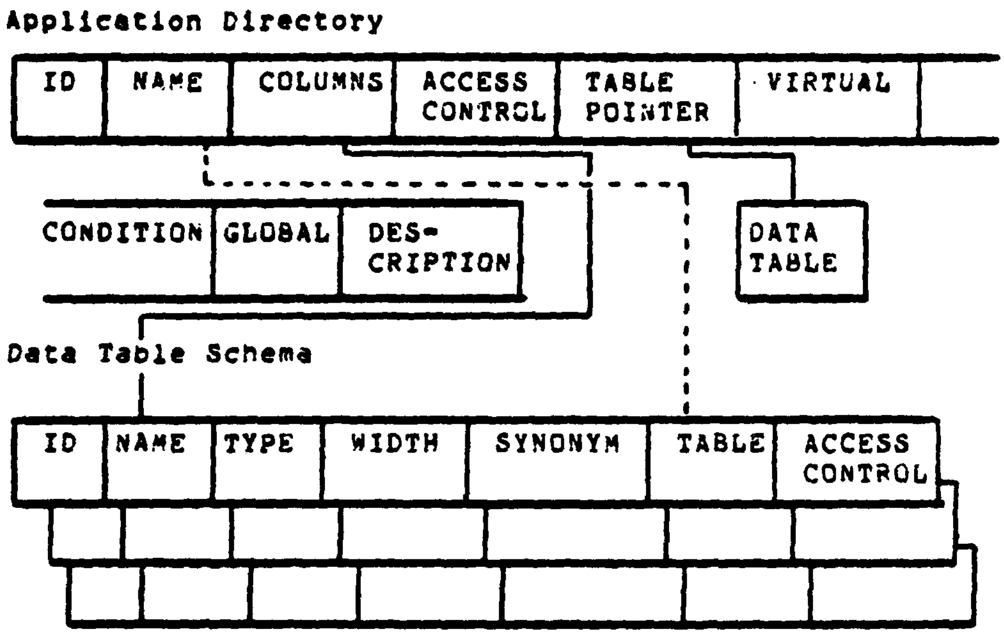


Figure 2.3 - IASS Table Relationships

This figure indicates that each row in the application directory table is connected to a set of rows in the data table schema table and a data table. The dotted line shows a cross reference from a data table schema table row to a directory table row.

1. Text Processor Tables

a. Text Directory Table

The text directory table contains a row for each text data table. The directory, Figure 2.4, has the standard application directory schema. Since the text data tables can be output to a visual medium, each data table has a global print format. This data is stored in the GLOBAL columns or, if necessary, in a table accessed via the data in the GLOBAL columns, and contains the page length, right and left margin, top and bottom margin, number of lines per page, page header, page footer, tab spacing, and line spacing.

ID	NAME	COLUMNS	ACCESS CONTROL	TABLE POINTER	VIRTUAL	CONDITION	GLOBAL	DES-CRIPTION

Figure 2.4 - Text Directory Table Schema

b. Text Data Table Schema Table

The text data table schema table, Figure 2.5, contains the predefined column set, ID and TEXT LINE. The ID is the primary key of the text data table. TEXT LINE describes the TEXT LINE column in the text data table. As

Figure 2.5 shows, the TEXT LINE can be aliased with the FORM LINE column in a form text data table or the BODY column in a mail data table.

ID	NAME	TYPE	WIDTH	SYNONYM	TABLE	ACCESS CONTROL
1	ID	IN- TEGER		NONE	ALL	read: all write: DBA modify: DBA
2	TEXT LINE	CHAR		FORM LINE, BODY	ALL	read: all write: all modify: DBA

Figure 2.5 - Text Data Table Schema Table

c. Text Data Table

The text data table, Figure 2.6, is described by the COLUMNS and DESCRIPTION columns in the text directory table. It contains data used to prepare a printed document or a computer program. The rows in a text table are sorted on the ID column. Each row has an unique ID number which corresponds to the line number in the display. Although the user can refer to an ID number, an ID number cannot be directly modified. The data in a TEXT LINE is unformatted. In a single line of text, there are two kinds of data that are recognized, the character string to be printed and special combinations of characters to be executed. The

executable characters are specific to a text processor application program (e.g. text formatter or compiler). Figure 2.6 shows the two kinds of data. Rows 1 and 2 contain literal character strings to be printed. Row n contains a formatting command (page-break).

ID	TEXT LINE
1	Now is the time
2	for all good men to
n	.pa

Figure 2.6 - Text Data Table

2. <u>Database Tables</u>

a. Database Directory Table

The database directory table contains a row for each database data table. The database directory table, Figure 2.7, has the standard application directory schema. The GLOBAL column contains data describing the display mode or printed format.

ID	NAME	COLUMNS	ACCESS CONTROL	TABLE POINTER	VIRTUAL	CONDITION	GLOBAL	DES-CRIPTION

Figure 2.7 - Database Directory Table Schema

b. Database Data Table Schema Table

The database data table schema table, Figure 2.8, initially contains an entry only for the ID column. The ID is a key in the database data table. As database data tables are defined, entries to the database data table schema table will have to be made.

ID	NAME	TYPE	WIDTH	SYNONYM	TABLE	ACCESS CONTROL
1	ID	INTEGER		NONE	ALL	read: all write: DBA modify: DBA

Figure 2.8 - Database Data Table Schema Table

c. Database Data Table

Each database data table, Figure 2.9, represents one entity, and is described by the COLUMNS and DESCRIPTION column in the database directory table. The data in a database data table is formatted. Each row is an unique occurrence of the entity. The columns of the database data table are the attributes of the entity. ID contains the display order of a row. This ordering does not imply that there is a canonical ordering of the entity. The ID can be referred to but cannot be directly modified by the user. The other columns of the database data table are not predefined. The database data table can be directly viewed

at the screen by the user or printed, but may be reformatted based on the data in the GLOBAL column if necessary.

ID	ATTR-1	ATTR-2	-	ATTR-n
1			-	
2			-	
-	-	-	-	-

Figure 2.9 - Database Data Table

3. <u>Form Generator Tables</u>

a. Form Text Directory Table

The form text directory table has the standard applications directory schema, Figure 2.10. Because the form is intended to be printed, each form contains a print format which is defined by the data aggregate named GLOBAL. This data aggregate is the same as that contained in the text directory table previously described.

ID	NAME	COLUMNS	ACCESS CONTROL	TABLE POINTER	VIRTUAL	CONDITION	GLOBAL	DES-CRIPTION

Figure 2.10 - Form Text Directory Table Schema

b. Form Text Data Table Schema Table

The form text data table schema table, Figure 2.11 contains the predefined column set ID and FORM LINE. The ID is the primary key of the form text data table. FORM LINE describes the FORM LINE column in the form text data table. Figure 2.11 shows that the FORM LINE column can be aliased with the TEXT LINE column in a text data table or the BODY column in mail data table.

ID	NAME	TYPE	WIDTH	SYNONYM	TABLE	ACCESS CONTROL
1	ID	IN-TEGER		NONE	ALL	read: all write: DBA modify: DBA
2	FORM LINE	CHAR		TEXT LINE, BODY	ALL	read: all write: all modify: DBA

Figure 2.11 - Form Text Data Table Schema Table

c. Form Text Data Table

A form is a repetitive document with blanks to be filled in. A form can be represented as a collection of rows contained in a table, as shown in Figure 2.12, and is described by the COLUMNS and DESCRIPTION columns in the form text directory table.

ID	FORM LINE
1	{NAME}
2	{ADDRESS}
3	
4	Dear {B.RELATION}
5	FORM1.TXT

Figure 2.12 - Form Text Table

Each row of the table is sorted on the ID and corresponds to the same row in the form. Although the user can reference an ID number, it cannot be directly modified. The FORM LINE column, contains unformatted data. In addition to literal character strings which are printed, including horizontal and vertical lines, it contains a special set of executable data. A special character combination indicates whether the blanks in the form are to filled in by the user and stored in a table, or whether the blanks are to be filled in from a table. This special set of data must be known by the user as it is merely inserted as a combination of characters into the FORM LINE column. The view of the form at design and modification time is exactly that of the form data table although it may be reformatted if necessary.

Figure 2.12 contains an example of the data in the form text data table. The data in the first FORM LINE indicates that the printed value for the first row in the

printed form is to be retrieved from the NAME column of the selected row of the associated database data table. The data in the fourth FORM LINE indicates that the printed form is to contain the literal string 'Dear' followed by the value from the RELATION column of table B of the associated database data table. The information in row 4 also indicates that the associated table is a join of two tables each containing a column named RELATION. Row 5 indicates that the body of the form is a text data table named FORM1.

A form text data table is similar to a text table. They differ only in their data content and the way they are used. The data in the form text data table can be variables whose values are determined at print time. The variables in a form do not have to be bound to a named data table or specific row in a data table and therefore can be re-used in the same or in many different applications.

4. <u>Electronic Mail Tables</u>

a. Mail Directory Table

ID	NAME	COLUMNS	ACCESS CONTROL	TABLE POINTER	VIRTUAL	

CONDITION	GLOBAL	DES-CRIPTION

Figure 2.13 - Mail Directory Table Schema

The mail directory table contains a row for each mail table. The standard application directory schema does not have to be modified for the mail application, Figure 2.13. The GLOBAL data items contain the data to determine ownership of the mail data table and any other applicable information.

b. Mail Data Table Schema Table

ID	NAME	TYPE	WIDTH	SYNONYM	TABLE	ACCESS CONTROL
1	ID	INTE-GER		NONE	ALL	read: all write: DBA modify: DBA
2	FROM	CHAR		NONE	ALL	read: all write: all modify: DBA
3	TO	CHAR		NONE	ALL	read: all write: all modify: DBA
4	COPY TO	CHAR		NONE	ALL	read: all write: all modify: DBA
5	DATE	CHAR		NONE	ALL	read: all write: all modify: DBA
6	SUBJ	CHAR		SUB-JECT	ALL	read: all write: all modify: DBA
7	BODY	CHAR		TEXT LINE, FORM LINE	ALL	read: all write: all modify: DBA

Figure 2.14 - Mail Data Table Schema Table

The mail data table schema table, Figure 2.14, contains the system defined columns of a message. The ID is the primary key of each mail table. The other columns contain the data to denote the originator, recipients, subject and the body of the message. Figure 2.14 shows that the BODY can be synonymous with the TEXT LINE of a text data table or FORM LINE of a form text data table

c. Mail Data Table

ID	FROM	TO	COPY TO	DATE	SUBJ	BODY
1	ME	YOU	THEM	1/1		MESSAGE
2	YOU	ME	THEM	1/2		MSG.TXT
~	~	~	~	~	~	~
n						

Figure 2.15 - Mail Data Table

Electronic mail is a utility which facilitates the exchange of textual messages between system users. A mail file contains a set of messages. The message consists of data items which are read by the recipients. The mail table, Figure 2.15, is described by the COLUMNS and DESCRIPTION columns in the mail directory table. Each row in the table corresponds to one message. Each message

contains a header which is comprised of the FROM, TO, COPY TO, DATE, and SUBJECT of the message. The BODY column can contain the entire message or the name of a text table. For example, Figure 2.15 shows that message 1 contains the entire message body. Message 2 contains the name of a text data table in the text directory table. The mail table can be directly viewed by the user as a summary of the messages but should be reformatted to read or edit a single message.

5. Electronic Spread Sheet Tables

a. Spread Sheet Directory Table

The spread sheet directory table contains a row for each spread sheet data table. The standard application directory schema, Figure 2.16, is used.

ID	NAME	COLUMNS	ACCESS CONTROL	TABLE POINTER	VIRTUAL	CONDITION	GLOBAL	DES-CRIPTION

Figure 2.16 - Spread Sheet Directory Table Schema

The GLOBAL data items define the recalculation order, default format, and any other information applicable to the referenced spread sheet data table.

b. Spread Sheet Data Table Schema Table

The spread sheet data table schema table, Figure 2.17, contains the predefined columns of the spread sheet data table. ID is the primary key of the spread sheet data table. The other columns contain the data to calculate the value of an entry position and construct the user view of the spread sheet data table.

ID	NAME	TYPE	WIDTH	SYNONYM	TABLE	ACCESS CONTROL
1	ID	IN-TEGER		NONE	ALL	read: all write: DBA modify: DBA
2	X INDEX	CHAR		NONE	ALL	read: all write: all modify: DBA
3	Y INDEX	IN-TEGER		NONE	ALL	read: all write: all modify: DBA
4	FORMAT	CHAR		NONE	ALL	read: all write: all modify: DBA
5	VALUE	CHAR		NONE	ALL	read: all write: all modify: DBA
6	FUNC-TION	CHAR		NONE	ALL	read: all write: all modify: DBA

Figure 2.17 - Spread Sheet Data Table Schema Table

c. Spread Sheet Data Table

A spread sheet data table contains the formatted data to calculate the values of and display a numerical model. The spread sheet data table is described by the COLUMNS and DESCRIPTION columns in the spread sheet directory table. A spread sheet data table, Figure 2.18, is pointed to by a TABLE POINTER in the spread sheet directory table.

ID	X	Y	FORMAT	VALUE	FUNCTION
1	A	1	INTEGER RIGHT 3	5	5
2	A	2	INTEGER RIGHT 3	5	5
~	~	~	~	~	~
n	A	3	INTEGER RIGHT 3	10	A1 + A2

Figure 2.18 - Spread Sheet Data Table

The spread sheet data table can be considered to be the tabular representation of the traditional spread sheet view, Figure 2.19, or, conversely, the traditional spread sheet view can be regarded as one display mode of the spread sheet data table. Each entry position in the view is defined by one row in the spread sheet data table.

	A	B	C	D	E	F
1	5					
2	5					
3	10					

Figure 2.19 - Spread Sheet View

The ID is the display order of the spread sheet data table but does not provide the ordering for the entry positions in the spread sheet view. The X and Y INDEX columns represent the relative position of the entry position in the spread sheet view. The mapping of the X and Y indices to the display is contained in the GLOBAL column of the spread sheet directory table. The FORMAT column contains the data describing the display of the entry position. A numeric value of an entry position can be displayed as an integer, floating point, or dollar and cents number. The format information also indicates whether the values should be right or left justified and the width of the entry position in the display. The VALUE column contains the display value of the FUNCTION column. The FUNCTION column contains an expression which is used to determine the value of the entry position. The expression could be the typical constant, literal, or arithmetic types, or could be a database query, or a pointer to any other IASS data table. The operands of an expression can be any constant value or the value of another entry position in the

spread sheet. An operand value can also be an arithmetic, trigonometric, or some other predefined function, which can use the value of another entry position as a parameter. Using the value of another entry position as an operand in an expression is necessary to support dynamic modeling, i.e. when a change is made to one entry position, it is immediately reflected in the entire spread sheet view. As a matter of fact, the freedom to define value of one entry position in terms of any other entry position resulted in one data table structure and it also prevented us from using the database integrity enforcement mechanism for the spread sheet view.

III. CONCEPTUAL INTEGRATION

A. OVERVIEW

The main design objective of the Integrated Application Software System (IASS) is to present the user with a single conceptual view of the system regardless of the context of its use. From the user's perspective, there is only one data object, the table. As was indicated in Chapter 2, depending on the level of experience or intent of the user, a translation may be required to reformat a data table. This translation is from table to table and therefore, the notion of a single data object is preserved.

At the conceptual level of each application there is a common set of table operators and a set of application specific table operators. The common set of table operators represents the transportable knowledge of the system as the user logically traverses between applications. The user must learn or be cognizant of only the application specific operations as the system use changes.

By functionally categorizing each data manipulation operations on the logical file of the non-DBMS applications, an intersection can be deduced. The intersection is comprised of operations to locate, insert, modify, delete, copy, and move data in a file. These operations can be

integrated at the conceptual level by a set of six basic IASS primitive table operators based on the relational algebra. This chapter demonstrates how the data manipulation operations of each selected application can be mapped into the conceptual level primitives. Although not the main intent of this chapter, where appropriate, extensions to the typical operation are suggested.

B. BASIC IASS PRIMITIVES

The six basic IASS primitives which can perform the operations in the functional intersection are INSERT, MODIFY, DELETE, PROJECT, SELECT, and UNION. Each primitive is set theoretic in that the operands are tables and the results are tables. A table can contain any number of rows. A special table, BLANK, is defined to be a row with all columns blank except for the ID. A literal string, 'literal', can stand for any character string in which it is contained. What follows is a description of the primitives. For this discussion the following conventions will be used:

(1) The word table is synonymous with data table

(2) Whenever two tables are used in an operation, table1 and table2, table1 will be the current table.

(3) Column names will appear in upper case, their value will be appear in lower case.

1. <u>Insert</u>

Given table1, INSERT adds table2 at a specified location. The operator is denoted:

INSERT(location, table2, table1)

2. <u>Modify</u>

Given a table, MODIFY changes the value of the columns in the rows of the table. The operator is denoted:

$$\text{MODIFY}((\text{COLUMN, column, new value})^{+}, \text{table})$$

where the 3-tuple (COLUMN, column, new value) describes the change by column name, present value, and the new value. The + indicates that more than one column can be modified by a single operation. If a change to a column value is to be made irrespective of the present value, e.g. change any value in column NAME to 'JONES', that desire can be expressed by a special character, or wild card, placed in the present value position of the 3-tuple.

3. <u>Delete</u>

Given a table, DELETE deletes the set of rows from the table that satisfy a specified condition based on the column values. This operator is denoted:

DELETE(condition, table)

The operands of the conditional statement are literal or numeric constants, arithmetic expressions, or the column

values of the table. The operators of the conditional statement are the arithmetic comparison operators (<, >, =, ≠, ≤, ≥), the logical operators (¬, ∧, ∨), and the arithmetic operators (+, -, *, /). The delete operator also creates a table that contains the deleted rows.

4. Project

Given a table, a projection of the table is made by removing some of its columns and/or rearranging some of the remaining columns. A projection of a table is denoted:

PROJECT(column list, table)

where column list names the desired columns from the table.

5. Select

Given a table, a selection returns the set of rows that satisfy a conditional statement based on the column values. A selection on the table is denoted :

SELECT(condition, table)

6. Union

Given two tables, table1 and table2, the union creates a table whose rows are in table1 or table2, or both. The union operation is denoted:

UNION(table2, table1)

The schema of the resultant table will be the same as table1. The columns in table2 whose content is not the same

as a column in table1 will not be in the resultant table. There are two differences between the union and insert operators. First, union appends table2 to the bottom of table 1. Second, insert assumes that tables 1 and 2 are the same type.

C. REALIZATION OF LOGICAL OPERATIONS

This section demonstrates that the data manipulation operations on the logical file in the functional intersection of each application can be expressed in terms of the conceptual level primitives. It will be assumed that the underlying system maintains the ID column as rows are moved in a table. Its resolution, therefore, will not be discussed.

1. Text Processor/Form Generator

The command sets of several text processors and the form generator facility of DBASE II and the ZIP form generator were reviewed, and it was found that the text processor set contained the set of form generator commands. Therefore, these applications will be discussed together.

a. Locate

Positioning the cursor typically comprises a large portion of the text processor and form generator operations. The cursor can be directed to a line number, relative distance from the current line, or to a substring. The result of positioning the cursor can be considered to be

a line reference. Locating a row in the text or form text data table is done by:

SELECT(condition, table)

Using the primitive, the text or form text data table can be browsed by contiguous lines (e.g $ID \geq X1 \wedge ID \leq X2$), or by line content (e.g. TEXT or FORM LINE = 'substring'$\wedge$(ID < id + 10)).

b. Insert

Inserting a row into a text or form text data table at location ID is done by:

INSERT(ID, BLANK, table)

The same primitive can be used to insert an entire data table2 into the current data table1 at location ID:

INSERT(ID, table2, table1)

c. Modify

Inserting and deleting characters are functionally equivalent in that they are modifications to the contents of a file. Assuming that the desired row is current, the operation to modify the row in a text or form text data table is:

MODIFY((TEXT or FORM LINE), 'old', 'new', table)

The find and replace operation is an extension of inserting

or deleting characters. With this operation, a row does not have to be previously selected. Also, a single change to a set of rows identified by a conditional expression based on their column values can be done. The find and replace operation is done by the expression:

```
MODIFY((TEXT or FORM LINE, 'old', 'new'),
     SELECT(condition, table))
```

d. Delete

Deleting a row or set of rows (block), from a text or form text data table is done by:

```
DELETE(condition, table)
```

The condition can be any function of the ID and/or TEXT or FORM LINE columns. This generalization enhances the typical text processor or form generator operation since rows can be identified by number or content and a block does not have to be a contiguous set of rows.

e. Copy

Copying lines in a text or form text data table can be done by the expression:

```
INSERT(ID,SELECT(condition, table), table)
```

Any portion of a text or form text data table1 can be copied to or saved to another table2 by the expressions:

```
UNION(SELECT(condition, table1), table2)
                 or
UNION(DELETE(condition, table1), table2)
```

The general nature of the primitives enhance the typical text processor or form generator operation by allowing the lines to be identified by content and not requiring that a block be contiguous. For example, the statement:

```
UNION(SELECT(TEXT LINE = 'ABC'∧ (ID < 10),
        table1), table2)
```

would copy to table2, any line in table1 with ID less than 10 and whose TEXT LINE column contains the character substring ABC.

f. Move

Moving rows or a block in a text or form text data table to location ID is done by the expression:

```
INSERT(ID, DELETE(condition, table), table)
```

The condition can be any function of the ID and/or TEXT or FORM LINE columns. This generalization enhances the typical text processor or form generator operation which requires that line numbers be known and a block of lines be contiguous.

2. <u>Electronic Mail</u>

The UNIX mail utility is an elaborate system which is closely coupled to the operating system. Viewed as a database table, the complexity is reduced. A set of

essential mail operations were deduced from the UNIX mail system.

a. Locate

Displaying messages for reading or editing can be done by:

SELECT(condition, table)

Using the primitives, the message to be displayed can be described by any condition of the columns of the mail data table. This would preclude the user from having to browse the mail data table first to determine which messages might be of interest and then listing them by number.

A summary of the messages can be displayed by selecting a set of messages which satisfy a condition and then displaying the desired columns:

PROJECT(column list, SELECT(condition, table))

By using the primitives, the user is not restricted to the predefined message summary.

b. Insert

A message can be created by appending a blank row into the mail data table and modifying its null contents:

```
UNION(BLANK, table)
MODIFY((COLUMN, Ø, 'new')+,
      SELECT( all columns = Ø, table))
```

Alternately, a message can be created at any location, ID, in the mail table by the expression:

```
INSERT(ID, BLANK, table)
MODIFY((COLUMN, Ø, 'new'),
        SELECT( all columns = Ø, table))
```

Regardless of the method used, the 3-tuple list in the modify operator contains a set of values for each column in the mail table except the ID. The change for each column is from null to the desired value supplied by the user.

Messages addressed to a user can be picked up from any mail data table2 by the expression:

```
UNION(DELETE(TO = 'user', table2), table1)
```

Messages can be picked up from a mail data table2 that satisfy any specified condition by the general expression:

```
UNION(DELETE(condition, table2), table1)
```

Finally, an entire mail table2 can be inserted into mail table1 at location ID by:

```
INSERT(ID, table2, table1)
```

c. Modify

Assuming that a message has been selected, any field in the message can be edited by the operation:

```
MODIFY((COLUMN, column, new value), table)
```

The same change can be made to several messages which satisfy a specified condition by slightly modifying the basic operation:

```
MODIFY((COLUMN, column, new value),
       SELECT(condition, table))
```

d. Delete

Messages can be deleted from a mail data table based on any column condition by the expression:

```
DELETE(condition, table)
```

Using the primitives, a user is not restricted to a predefined method of deleting messages. With one operation, any set of messages can be identified and deleted.

e. Copy

There may be an occasion when a copy of a message in a mail data table needs to be made. Copying a message can be done by:

```
UNION(SELECT(condition, table), table)
```

Messages can be copied or saved to another mail data table2 by the expressions:

```
UNION(SELECT(condition, table1), table2)
                  or
UNION(DELETE(condition, table1), table2)
```

f. Move

Messages in a mail data table are not ordered. They can be moved within a mail data table to location ID by the expression:

INSERT(ID,DELETE(condition,table), table)

3. Electronic Spread Sheet

As was noted in chapter 2, the typical spread sheet view is not the spread sheet data table. However, data manipulation operations on the view can be translated into the operations on the spread sheet data table.

a. Locate

Entry positions in the spread sheet view are referenced by X and Y position. As a result of the locate operation, the FUNCTION field is displayed. Locating entry position c, r in the table is done by the expression:

$$\text{PROJECT(FUNCTION, SELECT}(X = c \wedge Y = r, \text{table}))$$

The expression suggests that an entry position can be located by any other column in the spread sheet data table by the expression:

PROJECT(FUNCTION, SELECT(condition, table))

A problem with VISICALC is that a model cannot be debugged very easily since the only one entry position can be referenced at any time. The general nature of the primitive

operators enhances the debugging capability by allowing a set of entry positions to be located in the spread sheet data table by any condition of the column values. For example, all entry positions which have a VALUE greater than 10 can be found by the single operation:

```
SELECT(VALUE > 10, table)
```

This same action using the VISICALC command set would require the user to locate the entry positions one-by-one.

b. Insert

One row in the spread sheet view is composed of C rows in the spread sheet data table (C = number of columns in the view). Inserting one row in the spread sheet view at r, is done by C iterations of the expressions:

```
UNION(BLANK, table)
MODIFY(((X, Ø, x ),(Y, Ø, r)),
      SELECT (X = Ø ∧ Y = Ø, table))
```

In the expression, x is an element of the column set {1..C}.

One column in the spread sheet view is composed of R rows in the spread sheet data table (R = number of rows in the view). Inserting a column in the spread sheet view at d, is done by R iterations of the expressions:

```
UNION(BLANK, table)
MODIFY(((X,Ø,d),(Y,Ø,y)),
      SELECT(X = Ø ∧ Y = Ø, table))
```

In the expression, y is an element of the row set {1..R}.

Two spread sheets can be appended together to form one composite table by:

UNION(table2, table1)

Following a row or column insertion, further processing has to be done to move the successor rows or columns in the spread sheet view. Moving rows or columns is discussed in subsection f. After the rows or columns are moved, the entry positions that use the value of a moved entry position as an operand in the FUNCTION column must be found and modified. This process is described in subsection c. Finally, the modified FUNCTIONS must be evaluated and the new entry position VALUE displayed.

c. Modify

A value or label entry operation in VISICALC is done to change the value of the FUNCTION column of the current entry position. The expression is:

MODIFY((FUNCTION, function , new function), table)

Moving an entry position (X = e, Y = p), in the spread sheet view by changing the X or Y value (X = c, Y = r) may require a subsequent modification to the FUNCTION column of the entry positions that use e, p as an operand. The primitive expression to find and modify all of these dependent entry positions is:

```
MODIFY((FUNCTION, 'ep', 'cr'),
     SELECT(FUNCTION = 'ep', table))
```

The current entry position can be blanked by the expression:

```
MODIFY((FUNCTION, function, Ø), table)
```

Clearing all entry positions in the spread sheet view can be done as a special case of the blanking action previously described. Instead of the table being a previously selected row, it is in this case, the entire table:

```
MODIFY((FUNCTION, function, Ø), table)
```

After modifying the FUNCTION column of an entry position, the FUNCTION must be evaluated and the new entry position VALUE displayed.

The VISICALC format commands deal with the visual display of entry positions. Formatting an entry position requires a modification to the FORMAT column in the spread sheet data table. Changing the format of the current entry position is done by:

```
MODIFY((FORMAT, format, new format), table)
```

One format change can be made to a set of entry positions which satisfy a specified condition by the operation:

```
MODIFY((FORMAT, format, new format),
     SELECT(condition, table))
```

d. Delete

One row from the spread sheet view can be deleted with the VISICALC command set. Since one row, r, in the view is composed of C rows in the spread sheet data table, (C = total number of rows in the view) that many must be deleted from the spread sheet data table. The operation is:

$$DELETE(Y = r, table)$$

Although the VISICALC command set does not allow multiple rows to be deleted, the basic delete operation can be modified to delete a block (Y1 through Y2) in the spread sheet view:

$$DELETE(Y \geqslant Y1 \wedge Y \leqslant Y2, table)$$

Deleting a column is similar to deleting a row. Since a column, c, in the spread sheet view is composed of R rows (R = number of rows in the view) in the spread sheet data table, that many are deleted by the single expression:

$$DELETE(X = c, table)$$

This operation can be enhanced to allow the deletion of a set of columns (X1 through X2):

$$DELETE(X \geqslant X1 \wedge X \leqslant X2, table)$$

Following a row or column deletion, further processing is needed on the table. In the view, a deletion requires that all rows and columns be moved to fill in the blank. Moving rows and columns is discussed in subsection f. After the rows or columns are moved, all of the dependent tuples must be found and their FUNCTIONS modified to correspond to the new positions. Finally, the modified FUNCTIONS must be evaluated and the new entry position VALUES displayed.

e. Copy

Copying the current entry position (X = e, Y = p), to any other position (X = c, Y = r), in the spread sheet view, can be done by the expression:

```
MODIFY((COLUMN, column, (e,p).column)⁺,SELECT(X = c ∧
                                    Y = r, table))
```

If the destination entry position (X = c, Y = r) is not found, a new tuple must be entered into the spread sheet table and its null contents modified:

```
UNION(BLANK, table)
MODIFY((COLUMN, Ø, (e,p).column)⁺, SELECT(X = Ø ∧
                                    Y = Ø, table))
```

For the entry position (X = c, Y = r), the 3-tuple list contains a change for each column except X and Y, to the value of the same column in entry position at (X = e, Y = p). Although order in the spread sheet data table is

insignificant, the entry position can be copied to a specified location by substituting the insert operator for the union in the previous expression:

INSERT(ID, BLANK, table)

A column of height h, can be created in the view by making h copies of the entry position at X = e Y = p. In the spread sheet data table, the operation can be done by selecting the rows with a Y value in the height range (r through s) of the column, c, in the view, and making the same change to the VALUE, FUNCTION, and FORMAT columns. The operation is:

MODIFY({COLUMN, column, (e,p).column},SELECT(X = c ∧ (Y ⩾ r ∧ Y ⩽ s), table))

Copying a column of height h, can be done by making h calls to the copy one entry position process. On each call, the Y value for both the origin and target entry position is incremented by one.

Making n copies of one column can be done by making n calls to the copy one column process. On each call, the X value is changed.

Copying a row of length l, can be done by making l calls to the copy one entry position process. On each call, the X value for both origin and target entry position is incremented by one.

Making m copies of a row could be done by making m calls to the copy one row process. On each call, the Y value would be changed.

Any rectangular portion of a spread sheet view in spread sheet table1, (Y1 through Y2) by (X1 through X2), can be copied to or saved to another spread sheet in spread sheet table2. This operation is done by the expression:

$$\mathrm{UNION}(\mathrm{SELECT}((Y \geq Y1 \wedge Y \leq Y2) \wedge (X \geq X1 \wedge X \leq X2), \mathrm{table1}), \mathrm{table2})$$

or

$$\mathrm{UNION}(\mathrm{DELETE}((Y \geq Y1 \wedge Y \leq Y2) \wedge (X \geq X1 \wedge X < X2), \mathrm{table1}), \mathrm{table2})$$

f. Move

A row move, from r to s, in the spread sheet view can be done in the table by modifying the Y value of the C rows in the spread sheet data table. The expression is:

$$\mathrm{MODIFY}((Y, r, s), \mathrm{SELECT}(Y = r, \mathrm{table}))$$

A set of rows (r through s) can be moved by the same expression with a different with a different set of parameters:

$$\mathrm{MODIFY}((Y, y, y - |r - s|), \mathrm{SELECT}(Y \geq r \wedge Y \leq s, \mathrm{table}))$$

This expression is for the case where the set of rows is moved up. To move the set of rows down, the new value in the 3-tuple would have to be modified to $y + |r - s|$.

Moving a column in the spread sheet view is done with the same primitives. The process will not be reiterated except to mention that each occurrence of Y in the original expressions would have to be changed to X.

After moving rows or columns, dependent entry positions have to be found and their FUNCTION columns modified to reflect the new positions. Finally, the modified FUNCTIONS must be evaluated and the new entry position VALUE displayed in the view.

IV. IASS EXTENSIBILITY

A. COMBINING IASS TABLES

From the review of the commercial application systems, it is clear that the non-DBMS applications selected for this study provide functions to manipulate data in one logical file. Combining files, of the same type, can be done by appending files together or inserting one into another. As a result of these file combinations, however, no new relationships are developed nor can information be deduced from the action. Using the relational database model as the common data model, there are a set of binary operators which can be used to combine tables to form new relationships and derive information. These operators UNION, SET DIFFERENCE, INTERSECTION, JOIN, and NATURAL JOIN are defined in Ullman [Ref. 2]. This chapter explores the semantics of combining the data tables by these operators. Speculation of this nature can result in numerous table combinations which could potentially define a new application. This review is not presumed to be exhaustive, but merely suggests the meaningfulness of and potential uses for the IASS table combining operators.

B. INTRA TYPE COMBINATIONS

This section considers the effect of combining data tables of the same type by the operators SET DIFFERENCE, INTERSECTION, JOIN, and NATURAL JOIN. The UNION operator will not be discussed since each application can use it to support an existing function.

1. <u>Text/Form</u>

Due to the similarities in the text and form tables, the semantics of the intra table combinations will be discussed together.

a. Set Difference

The set difference operator would be meaningful within the context of text processing and form generating. It could be used in applications which require a line-by-line comparison between two tables. For example, it is often necessary to compare two versions of the same computer program in the course of program development or two versions of the same form during design. By applying the set difference operator on two tables R and S, a listing of all the lines in R that are not duplicated in S would be returned.

The set difference operator would also be useful in an application to extract entire sections from a table. Used in this way, the operator would represent the inverse of the union or insertion operators to build a composite table. From the composite table R, those same sections, S,

could be directly removed by performing a set difference. Similarly, the set difference could also be useful to remove lines from a table, R, that were contained in table, S.

b. Intersection

The intersection of two tables would be meaningful within the context of text processing or form generating. It could be used in applications requiring a line-by-line comparison or to match substring patterns of two tables to determine their similarities. For example, comparing versions of the same table to check their consistency could be done by taking the intersection. This operator also suggests that two unrelated tables could be compared to determine their "closeness". The resultant table could be used to deduce similarities between the two tables based on the fact that they contained identical lines, or used to selectively remove duplicate lines from either table.

c. Join

Joining two text tables or form text tables would be meaningful. It could be used in an application which required two tables to be in context simultaneously. For example, tables equijoined on their ID number would produce a split-screen effect to review and edit them side-by-side. This combination would be particularly useful if the contents of one table was dependent on or related to

the contents of the other. The same operation would be a method to produce a multi-columned table from two tables.

Joining two tables would also be useful in applications which required a comparison of the lines of two tables. Because the join performs a cartesian product it would be unlikely that lines could be compared other than as to their equality. Since the join uses a selection operator, the join could determine equality by matching a line in R as a substring in S or vice versa. An equijoin on the contents in two tables would produce a table which would list the lines in R next to the lines S that were equal. This could be used to determine the similarity between two tables with finer resolution than that available by taking the intersection.

d. Natural Join

To do a natural join of text tables or form tables would be meaningful. A natural join between two tables would produce similar results as those obtained by doing an intersection or an equijoin on the table contents. An application in which the natural join could be used would again be to produce a table based on the equality of lines contained in two separate tables. Instead of creating a table which contained a line from each table in one row, the resultant table would contain only lines from one. As with the equijoin, the lines returned could be a substring match.

2. Mail

a. Set Difference

The set difference operator would be meaningful in a mail utility. An application in which the operation would be useful is to eliminate duplicate messages from several mail tables.

b. Intersection

The intersection of two mail tables would be meaningful. By doing an intersection, duplicate messages could be located in several mail tables. This information could be used to selectively manage the message tables and control the number of message copies in the entire system.

c. Join

The join of two mail tables would be useful in a mail utility. It could be used in an application such as automatic readdressing. For example, consider the set of messages that have the same subject. A recipient could be in mail table R by virtue of the fact that it has received at least one message pertaining to the subject. This mail table in effect would represent a channel defined by the common subject. As messages are received in the mail table S, an equijoin on the SUBJECT column between mail table R and mail table S followed by a projection of R.header and S.Body would create a copy of the new message for each recipient in mail table.

The join of two mail tables would also be meaningful in an ad hoc application to find messages in two mail tables whose fields have a specified relationship. For example, a join could be done to return all of the messages in R and S such that they were from the same addressee but the messages in S were dated after the messages in R.

d. Natural Join

A natural join on two mail tables would be meaningful. Since the BODY column of a message is textual, two messages could be considered to be equal if one was a substring match of the other. A natural join on two mail tables R and S would therefore, return all messages in R which had the same header as a message in S and whose body was either duplicated, a subset of, or a superset of the body of the message in S.

3. <u>Spread Sheet</u>

a. Set Difference

The set difference operator would be meaningful in a spread sheet application. It could be used to compare two instances of the same model. For example, if spread sheet R contained a model with one set of parameters, and spread sheet S contained the same model with a different set of parameters, the set difference would produce a spread sheet view which showed each entry position in R that was different in S.

b. Intersection

An intersection of two spread sheet tables would be meaningful. It could be used to produce a spread sheet used to compare different instances of the same model. The resultant table in this application would show the entry positions that remained constant given a different set of parameters.

c. Join

The join of two spread sheet tables would be meaningful. If the spread sheets were the same model with different parameters, an equijoin on the position fields would be a way to produce a table so that two spread sheets could be compared side-by-side. A process could be developed which could be used to toggle between the spread sheet in view. In this way, each spread sheet maintains it logical independence.

A join between two spread sheet tables would be the only way their contents could be compared by column relationships. For example, a join could be done to directly determine the differences between two instances of the same model or the same instance of a problem in two different models. The join on R and S such that the X and Y positions were the same but the VALUE in R was in a specified relationship to the VALUE in S would return a table which contained the entry positions in R and S which satisfied the condition.

d. Natural Join

A natural join on two spread sheet tables would produce the same effect as the intersection.

C. INTER TYPE COMBINATIONS

This section considers the effects of combining data tables of different types by the operators UNION, SET DIFFERENCE, INTERSECTION, JOIN, and NATURAL JOIN. Some table operations are not syntactically feasible on certain table types and therefore are not addressed.

1. <u>Text</u>

a. Union

The union between a text table and a form table would be meaningful. For example, the body of a letter can be kept in a text table. By unioning it to a form table containing a letter head, a form letter would be created.

The union between a text table and mail table would be meaningful. An application in which this operation would be useful to create a text table from the bodies of several messages in a mail data table.

The union between a text table and a spread sheet table would be meaningful. By appending a spread sheet table onto a text table, the FUNCTION column data could be included in a text table, possibly to be sent to an individual in a letter.

The union between a text table and a database table would be meaningful. If a database table contained a textual column, for example a literal description of an object, the data in that column could be included in a text table. Conversely, a textual description about an object could be kept in a text table. By unioning the text table onto the database table, the data for the textual column would be provided.

b. Set Difference

The set difference between a text table and a form table would be meaningful. An application to remove text lines from a text table which were also in a form could use the set difference. A set difference between a text table and any of the other table types would not be syntactically meaningful.

c. Intersection

The intersection between a text table and a form table would be meaningful in an application to determine the text lines that were common in the tables. By this operation it could be determined if a form letter contained the a body stored in the text table. An intersection between a text table and any of the other table types would not be syntactically meaningful.

d. Join

A join between a text table and a form table would be meaningful. A text table can be joined with a form

table on the ID fields which could be used to produce a form with a textual description. Either portion of this composite table could be separately edited. An equijoin on the lines of the text and form tables would return the lines which were common in both tables.

A join between a text table and a mail table would be meaningful. If the text table had one subject on each TEXT LINE, by doing an equijoin on the TEXT LINE and the message SUBJECT, a text table can be created which contains the message bodies pertaining to a set of subjects which are of interest. This could be a method of collating the messages from several system users concerning a particular subject into a single document.

A join between a text table and a database table would be meaningful. For example, if a database table contained a textual column, the column in the database table could contain the ID of a text line. An equijoin on the database column and text line would supply the text for the database tuple. One text line could be contained in several of the database tuples and by maintaining one copy of the textual contents, all tuples will be assured of having the same textual column value. If the database table contained a mailing list, a join on the database table and the text table would be the procedure by which a copy of the text table could be made for each entry in the list. Another application in which a join would be meaningful between a

text table and database table is where the database contains a set of keywords or key phrases. By doing an equijoin on the TEXT LINE and the keywords or key phrases, every line in the text table containing the keywords or key phrases would be returned. The same application also suggests that the combination could be used in support of a word checking program. Separate dictionaries can be maintained in a database table and joined with a text table. The resultant tables could be used to check spelling or to analyze a particular style of writing.

A join between a text table and a spread sheet table would be meaningful. A narrative description about one model could be maintained in a text table. Since it is common to store the same model with different parameters in several spread sheets, one text table could be joined to the spread sheets by ID number to document the model. This would be useful to the user viewing the spread sheet table. To contain this additional information in the view, would require an application specific process which would display the additional field.

e. Natural Join

The natural join between a text table and form text table would be meaningful since the column names are synonymous. It could be used to determine the lines of text contained in both tables. The resultant table would be the same as that returned by taking the intersection.

2. Form

a. Union

The union between a form text table and a text table would be meaningful. An application in which this operation would be useful would be to generate a form letter as was addressed in subsection 1. The union between a form text table and a mail table would be meaningful. The body of a message could be a FORM LINE. By unioning the form table and the mail table, a form sent line-by-line through the mail facility could be regenerated.

b. Set Difference

The set difference between a form text table and text table would be meaningful. In a similar application as that discussed in subsection 1, the operation could be useful to remove a set of text lines from a form. A set difference between a form text table and the other table types is not syntactically feasible.

c. Intersection

The intersection between a form text table and a text table would be meaningful to determine the common lines of text between the two tables as was described in subection 1. An intersection between a form text table and the other table types is not syntactically feasible.

d. Join

A join between a form text table and text table would be meaningful in applications discussed in subsection

1. A join between a form text table and a mail table would be meaningful. A user can sequence a FORM LINES and name the form in the SUBJECT column and send a form line-by-line through the mail facility. A form text table with one line can be created containing the form name. By an equijoin on the FORM LINE and SUBJECT columns, all of the FORM LINES could be collected from the mail table. The form can be resequenced by the data in the SUBJECT column of each message.

A join between a form text table and database table would be meaningful. A join on the ID numbers between a form text table and an associated database table would be useful to view the two tables simultaneously. The lines of each table in the joined table could also be edited independently in this form.

A join between a form text table and spread sheet table would be meaningful. Since the FUNCTION column could contain a reference to an entry in a database table, a form text table and spread sheet table could have a relationship through a common database. An equijoin on the FUNCTION column and FORM LINE would return the list of entry positions and form lines which contained the same database reference. In the view, the spread sheet and the filled out form could be displayed. This would be useful to show a spread sheet model and its parameter set in a form in one view.

e. Natural Join

A natural join between a form text table and text table would be meaningful in the same application described in subsection 1.

3. Mail

a. Union

The union between a mail table and text table would be meaningful. A message can be created by a union between a mail table and text table that contained a message body on one line. The message header can then be edited for the message created. Similarly, a text table of n lines could be unioned onto a mail table and sent to a system user. FROM, TO, COPY TO, and DATE columns of n messages would be the same. The text table name and ID can be placed in the SUBJECT column to direct the recipient in the reconstruction of the text table.

The union between the mail table and a database table containing a textual column is meaningful. The union would be useful to supply a message body from the textual column. The header of the message could then be edited. This would be a method to send data in a local database to any other user in the system. Another application for a union between a database and mail table would be to generate a set of message headers in a database table. These headers could be unioned onto a mail table and then the BODY columns provided.

b. Join

A join between a mail table and the form text table and text table is meaningful. For example, this operation can be used to send a text table or form text table to a system user through the mail facility. A set of message headers addressed to the same recipient containing an ID in the BODY column, could be prepared in the mail table. By doing an equijoin on the message BODY and the text or form text ID columns, and removing the extraneous columns, a message containing each line of the form text or text table can be created.

A join between a mail table and a database table would be meaningful. For example, a message could be addressed to a group of individuals recognized by a single name. A database table could contain the mapping from that single name to the individual names. An equijoin between the TO column of the message and the column containing the aggregate name in the database would produce a table which contains a copy of the original message for each individual included in the group. This operation could also be used to generate a message. Having prepared the body of a message and using a standard subject line, a database table containing a set of headers including the subject, could be equijoined with a mail table on the SUBJECT columns to produce the entire message. This message generation method is particularly useful in a situation where the standard

headers are always being revised. It is also useful when one message needs to be sent to several different headers or several messages of the same subject need to be sent to the same header.

Finally, a message can be created by joining a database table containing a set of message headers and a text table containing a message body on each line. By equijoining the two tables on the ID column, one heading would be joined to one body. A join on any other column in the header, and the TEXT LINE would join every header with every body.

c. Natural Join

The natural join between a mail table and database table would be meaningful. An application in which a natural join would be useful would be to support routing of incoming messages. For example, it is often necessary to route messages to individuals based on the message subject. If a database were maintained which had fields for the subject and name of a person, the mail table could be naturally joined to this list. The result of the action would be a table which contained a copy of the message for each person.

4. Spread Sheet

a. Union

The union between a spread sheet table and a database table would be meaningful. This operation could be

useful in a situation where a spread sheet must contain standard entries. For example, a database table could be maintained with a subset of the spread sheet table columns (e.g. X, Y, and FUNCTION). This database could be unioned onto a spread sheet table to boiler plate the spread sheet view.

b. Join

The join between a spread sheet table and database table would be meaningful. For example, a database table could contain a set of values which are of special significance. A conditional join based on these values and the VALUE column in the spread sheet table would return all of the entry positions which satisfied the condition. Another application in which this combination would be meaningful would be to store standard entry position definitions in a database table. An equijoin on the position columns would boiler plate the spread sheet. Finally, an application in which this combination would be meaningful is to store a set of parameters for a spread sheet table in a database containing columns for X, Y, and a vector containing the parameter set. By doing an equijoin on the X and Y columns, and removing all of the columns except the X, Y, FORMAT, VALUE, and desired parameter column, the parameter set is supplied to the model.

c. Natural Join

The natural join between the spread sheet table and a database table would be meaningful. A database table could contain documentation with respect to each entry position in the spread sheet table. By doing a natural join on the X Y columns, a spread sheet table could be documented tuple by tuple. This would especially be useful if there were instances of the same model in separate spread sheet tables. For this application, the line documentation would only have to maintained in one database table.

A spread sheet can be created from two database tables. Database table1 could contain the X, Y, and FORMAT columns representing a particular spread sheet format. Database table2 could contain the X, Y, VALUE, and FUNCTION columns representing a standard model. By a natural join of the two tables, a spread sheet containing a standard model in a selected format would be produced.

V. CONCLUSION

A. FINDINGS

Based on this study, it can be concluded that the relational database model can conceptually support the data representation and manipulation requirements of the selected IASS applications. At the conceptual level each logical file can be represented as a table and the common data manipulation functions of each application in its traditional form can be expressed in terms of basic IASS primitives. In the IASS, higher level application specific functions can be defined in terms of the lower level primitives. For example, the COPY, MOVE, and FIND AND REPLACE functions can be expressed in terms of the primitive pairs, INSERT/SELECT, INSERT/DELETE, and MODIFY/SELECT respectively. In addition to the intersection of commands and functions that exists between the IASS applications there exists an intersection of commands and functions between subsets of the applications. For example, formatting commands are applicable if a user is editing a text file, form, or message body. Also, aggregate arithmetic functions are common to database and spread sheet applications. These intersections further reduce the set of application specific commands and functions a user must know.

The IASS also contains a set of four combining operators. When used to combine tables of the same type, these operators can be used to deduce information about data contained in two tables, and to create new tables from existing tables. In this way, the IASS enhances the basic capabilities of each of the non-DBMS applications.

By mapping the logical file into one conceptual data object, data independence is achieved. Therefore, the full use of the specifically designed data tables can be realized by removing their semantic identity. A data table can be used by a logically different application or it can be combined with a different table type into a table which defines a new relationship. This new table can be created for an ad hoc application or for an application added to the IASS. The ability to combine data tables imparts to the IASS, capabilities which are not available from the set of disjoint applications.

B. FOLLOW-ON RESEARCH

The first task is to re-evaluate the logical data bases designed for the IASS. The present organization implies that each application is a disjoint database. This perspective was useful for this study, but it is clear that they must not be disjoint if it implies that different table types cannot be combined. Another iteration on the IASS tables needs to be done to combine the application

directories and data table schemas into tables in a centrally maintained data dictionary/directory.

After the conceptual level is re-evaluated, the next step in the project should be to design the physical level of the IASS and design the software to implement the system. It is recommended that the first iteration of the implementation be a prototype consisting of the table data object and the ten IASS primitives, to evaluate the utility of the IASS to support the needs of a user in an actual operating environment. Subsequent iterations can include the DBMS functions such as data integrity, security, and crash recovery.

Concurrent with the physical level and software design, the application specific command languages and processing programs can be designed. The text formatter, form printer, and spread sheet view generator can all be designed based on the data table definitions and the abstract interfaces of the primitives.

Finally, it is recommended that the IASS be emulated on an existing relational DBMS (e.g. DBASE II). The reason for this is two-fold. First it would provide an available test bed which can be used to test concepts which need to be resolved before the prototype system is delivered. Second, it could be used to determine the manner in which the IASS will handle the fundamental needs of the user before the

first prototype is finished. It would therefore, be a way to involve the user in the early design stages of the IASS.

LIST OF REFERENCES

1. Martin, J., Computer Data-Base Organization, Prentice Hall, 1975

2. Ullman, J., Principles of Database Systems, Computer Science Press, 1980

APPENDIX A: WORD STAR

WORD STAR is a word processing program developed by Micro-Pro to combine the capabilities of a screen editor and an on-screen text formatter. The result is a very powerful text editor which displays the referenced file as it will appear on the printed page.

WORD STAR is primarily menu-driven. The commands which are presently valid are displayed in a menu, and are executed by keystroke combinations. On-line information is available to the user concerning many other aspects of WORD STAR. The menu driven feature eases user initiation to WORD STAR and is part of the Help facility. The level of help is selectable to match the users level of experience, and determines the extent to which the menus are displayed on the CRT.

WORD STAR is composed of a set of seven hierarchically organized menus or environments, as shown in Table A.1. The user enters WORD STAR in the No-File environment. At this point there is no file in reference, the object granularity is the file, and the menu options include commands to: change the logged disk drive, set the automatic directory display feature (on/off), set the help level, print a file, rename a file, copy a file, delete a file, run a program, open a document file, and open a non-document file.

Table A.1 - WORD STAR Menu Hierarchy.

LEVEL	MENU
1	No File
2	Main Menu
3	a. Help b. On-Screen Format c. Print Control d. Quick Edit e. File/Block

WORD STAR recognizes two types of files, "document" and "non-document". A document file can either be a text file processed by a word processor or a program run by a computer. A non-document file is a special purpose file which is used by another software product, and will not be discussed further.

The on-screen editor and formatter are invoked by selecting the menu option to open a document file. This causes WORD STAR to enter the Main Menu environment with a specific file in reference. If the file previously existed it is made current, otherwise a new file is created and made current. On entering the Main Menu environment, a status line and a rule are initialized. The status line contains information about the system - the name of the file, the page within the file, the column and row number the cursor

is at, and the insertion mode (on/off). The rule indicates the right and left margin position as well as the tab positions. The Main Menu represents the basic file editing environment where the user will remain until it is decided to quit the current file and return to the No File Menu or the operating system. In any case, WORD STAR does not permit lateral movement between the sub-menus of the Main Menu.

A useful feature WORD STAR employs is "word wrap". With word wrap, the user does not have to insert carriage returns at the end of each line. As the text overruns the end of the line, WORD STAR automatically starts the next line. In this way, the user merely inputs an entire block of text as a continuous ASCII character string, and leaves the formatting to the system. In the Main Menu, the user can edit the file in granularities of character, word, and line. Insertion is a "toggled" operation (on/off), where the user is either in insert mode or overwrite mode. Any keystroke entered is either inserted in the text at the cursor position, shifting characters to the right to accommodate it, or overwrites the character at the cursor position. To facilitate on-screen editing, the Main Menu contains commands to control cursor movement and to scroll the screen. It is possible to insert tabs or end-of-paragraph markers. There is a "Find and Replace" command which can be repeated any number of times. Deletions can be done on a

single character, a word, or an entire line. The Main Menu also contains options to select one of the five submenus.

The Quick Editing environment supports editing on higher levels of abstraction of text objects than the Main Menu. There are additional cursor movement commands to give a wider range of control and granularity. As in the Main Menu environment, the user can scroll the display, but now it is continuous at nine user selectable rates until stopped by command. Insertions are accomplished in the same way as in the Main Menu environment, but deletions are possible on a wider range of objects. There is a feature to allow a command to be repeated at one of nine user selectable rates, until stopped by command.

The Block environment provides the user a set of operations on a block of text. WORD STAR considers an entire file to be a special case of a block of text. Files can be saved by several menu options: save and resume the referenced file, save and quit to the operating system, save and exit the referenced file, and copy to another file. Files may also be renamed, deleted, printed, or quit without saving changes. To support these file operations, the Block Menu contains options to change the logged disk, and to turn the automatic directory listing on or off. In this capacity, the Block environment is used as a successor to the Main or Quick Editing environments after the cursor is positioned. Blocks in a file must be marked by the user.

As a delimited aggregation of text, a block can be moved within the same file. Copying blocks of text can either be within the referenced file or between the referenced file and an external file. Block copying between files are bi-directional. Copying a block to an external file entails overwriting an existing file or creating a new file. Copying a block from an external file entails moving the entire external file to the point in the text indicated by the cursor. Any marked block can also be deleted. As a precautionary measure, WORD STAR allows the user to hide block markers, and only blocks which are visibly marked can be deleted. In addition to a text block being organized into a continuous, unstructured string of text, WORD STAR supports a columnar organization.

The previously described menus contain operations to create, edit, position the cursor, or output a text file. The format of the file, either as it is visually displayed or printed out, is defined by a set of formatting parameters associated with the file or by commands embedded in the file. The formatting parameters associated with a file are initially set to default values and the set of embedded commands is initially empty.

Formatting in WORD STAR is primarily done on-screen with the options contained in the On-Screen Menu. The on-screen formatting commands are those whose effects can be visually displayed, and they are listed in Table A.2.

Table A.2 - WORD STAR On-Screen Formatting Commands.

1. Set left margin
2. Set right margin
3. Release margins
4. Set and clear tabs
5. Indent a paragraph
6. Create a special rule
7. Center text
8. Set line spacing

The On-Screen Menu also contains options in the form of (On/Off) toggles to control: word wrap, rule display, variable tabbing, hyphenation help, right margin justification, soft hyphen, print embedded control characters, and page break display. If an on-screen formatting operation needs to be applied to the previous contents of the file, the applicable portion of the file must be reformatted. Furthermore, these formatting parameters are only temporarily applied when the file is referenced. Any subsequent reference to a file requires that the on-screen formatting parameters be reset.

WORD STAR combines into one menu, the Print Menu, all options which create special printing effects not normally displayable on a video screen. There are options to: bold face, double strike, underline, strike out, subscript, and superscript. Since the effects of these options cannot be

displayed on the video screen, a special character is used to mark the affected area. Additional special printing effects are selectable through this menu on a one time basis: overprint a character, indicate a non-break space, and overprint a line. The Print Menu also contains options which control the printer during output. The user may embed commands in the text file to cause the printer to change pitch, or cause a pause to allow the user to change the print element or ribbon.

Printing can also be directed through the use of embedded dot commands. These commands are placed in the text file and appear as regular text on the display, but are not output to a printer and force WORD STAR to change a printing parameter at print time. Dot commands alter the default parameters WORD STAR uses to format the printed page. Table A.3 provides a listing of these commands.

Dot-commands may be placed anywhere in the text, but since they are static and tend to destroy the relationship between what is displayed and what is printed, they are usually placed at the beginning of the text file. As with the options of the Print Menu, dot-command actions must be supported by the specific printer in use.

The last menu to be described is the Help Menu. Help is "on-line" in that it can be invoked at any time through the Main Menu, and is "dynamic" in that the level of help can be adjusted. The level will determine how much information is

displayed when an option is selected. The Help Menu options display information on: paragraph reforming, flags in the right-hand margin, dot and print commands, status line, ruler line, how to set margins and tabs, and how to move blocks of text.

Table A.3 - WORD STAR Dot Commands.

1.	Set line height
2.	Set page length
3.	Set top margin
4.	Set bottom margin
5.	Generate headers
6.	Generate footers
7.	Set footer margin
8.	Reset page number
9.	Offset page from left side of printer
10.	Position page number
11.	Set character width
12.	Force a page break
13.	Prevent a page break

WORD STAR is an excellent and very popular word processing program. The screen-oriented and on-line formatting features are different from other systems in that they are extremely easy to use. Once experience is gained with WORD STAR it is difficult to use line-oriented editors or off-line formatting systems. The on-line help facility makes WORD STAR easy to learn and user friendly. One aspect of WORD STAR that could be considered a disadvantage is the

large command set. However, being menu-driven, the commands not normally used do not have to be memorized since they are always listed in the menu.

APPENDIX B: VI

"VI" is a text editor used by the UNIX operating system and was created by the University of California at Berkeley, and Bell Laboratories.

VI (visual) is a display oriented interactive text editor with a command vocabulary size of about ninety one. The user sees the CRT screen as a window into the text file and all editing operations are immediately visible. Line numbers are not displayed and have no real use in VI, although it is possible to find out the number for a line. For the sake of protection the user does not actually edit the file, but a copy of it. At the completion of a session the user will indicate whether to keep the edited copy or the original.

There are forty seven movement commands for control of the cursor, which is the editor's point of reference, and the screen display. Scope of movement is possible over file, screen, paragraph, section, sentence, line, word, and character sized units. Up to twenty six locations in the file can be marked for later return, or specific locations found that match a desired character string. Table B.1 lists the cursor movement commands available in the VI system. Note that there is duplication, in that more than one command does the same thing.

Table B.1 - VI Cursor Movement Commands

1. Backward window
2. Forward window
3. Scroll down *
4. Scroll up *
5. Backspace one character *
6. Backspace a single character
7. Backup a word
8. Backup a word during insert
9. Backup to beginning of word
10. Retreat to previous line *
11. Retreat to beginning of sentence
12. Retreat to beginning of previous paragraph
13. Retreat to previous section boundary
14. Linefeed advance to next line
15. Advance to first non-white space on next line *
16. Advance to next line, first white space
17. Advance to next line, same column *
18. Advance to next character *
19. Advance to beginning of word
20. Advance to end of next word
21. Advance to section boundary
22. Advance to the next typed character
23. Advance to beginning of next paragraph
24. Move to previous line *
25. Move to end of current line *
26. Move to balancing parenthesis or brace
27. Moves cursor to last line on screen *
28. Moves cursor to middle of screen *
29. Move forward to beginning of word
30. Move forward to end of word
31. Move to first non-white space on current line
32. Move to line number # *
33. Search for word *
34. Search forward for string **
35. Search backward for string *
36. Search for next match **
37. Repeat last single character search
38. Find a single character, backwards *
39. Find a single character, forward *
40. Reverse direction of previous find

Table B.1 - (Cont.)

41.	Find first instance of next character
42.	Repeat the last search command *
43.	Homes the cursor
44.	Mark the present position of the cursor *
45.	Return to marked position *
46.	Redraw the screen
47.	Returns to previous context

The operations of insertion, modification and deletion are supported by thirty commands that permit the user a varied level of object control. Items that are inserted, modified or deleted are immediately updated on the screen to give the user a current view of the file status. The user also has the ability to undo the previous command if its effects were undesired. Most insertion and modification commands are structured so that they continue to operate until the user issues a command to terminate them. Normally during insertion the user has control of format in that new lines are started by entering a carriage return. However there is an option that will let VI determine when to start a new line, based on line length, and let the user just enter text as a continuous stream. Table B.2 lists the thirty edit commands.

In order to use VI the user issues the command "vi" followed by the name of the file to be edited. If this is a

new file, then the name will not be found in the directory and VI will create an empty file. After entry, the user will issue cursor motion commands to maneuver through the file, and issue edit commands to change the contents of the file. There are no other modes or displays available in VI.

Table B.2 - VI Edit Command Summary

1. Insert a number of spaces
2. Insert nonprintable characters
3. Insert "shiftwidth" blank spaces
4. Insert at the beginning of line
5. Insert at end of line
6. Insert before the cursor **
7. Insert after the cursor **
8. Insert new line below current line
9. Insert new line above current line
10. Insert text below current line **
11. Insert text above current line **
12. Delete last character
13. Delete rest of the text on current line *
14. Delete character before cursor
15. Delete the following object
16. Delete single character under cursor **
17. Repeat last command **
18. Join together lines *
19. Replace single character under cursor
20. Replace characters at cursor **
21. Change the entire line
22. Change single character
23. Change the following object
24. Change rest of the text on current line
25. Undo last change to current buffer **
26. Restore current line to previous condition
27. Yank following object into buffer *
28. Yank a copy of current line into buffer
29. Repeat last text insertion
30. Named buffer specification follows *

In addition to the two command categories already given there are additional commands of a miscellaneous nature. Table B.3 lists these additional commands.

Table B.3 - Miscellaneous VI Commands.

1.	Print file status message
2.	Clear and redraw the screen
3.	Redraw the current "logical" screen
4.	Suspend or restart output
5.	Cancel partially formed command
6.	Return to position in last edited file
7.	Reformat lines in buffer
8.	Indicate file and option manipulation
9.	Quit VI, enter line-oriented editor

Some very basic formating commands for line length and indenting are directly available. A macro creation capability is present to allow the user to create abbreviations for command strings. Table B.4 lists these formatting commands. VI makes no claim to supporting a formatting package, since the file will be output in the same format the user entered it. For special formatted output a VI generated file must be processed by an off-line word processor, like "NROFF -ME" described in Appendix (D).

VI provides a high degree of support to the user for restructuring a file, or files. There are nine buffers available for storing deleted text, and twenty six buffers

to use as temporary holding spaces while reordering and editing. The text can be taken from other files and/or buffers, for use in the file currently being edited. If needed, previously deleted text from the current file can be recovered, and also other files.

Table B.4 - VI Formatting Commands.

1.	Reformatting command
2.	Shift lines left one "shiftwidth"
3.	Reindent lines
4.	Shift lines right one "shiftwidth"
5.	Prints current file contents

"VI" is a good screen oriented editor and has a wide range of capabilities, however it has some drawbacks.

(1) It has a poorly designed user interface since the command vocabulary is very large and the individual command strings are difficult to remember. There does not seem to have been much thought given to the design of the command vocabulary.

(2) It takes a fairly long time to learn the VI system and gain functional use. An on-line tutorial program is used to help beginners, since it is hard to become familiar with it on their own.

(3) VI does not inspire user confidence in that it is too easy to accidentally enter some unknown command string, and there is little correlation between what the user wants to do and the command(s) that must be issued.

(4) From personal use, about thirty three commands were considered to be generally useful (marked by * or **), and only ten of these accounted for the greater majority of all operations (marked by **). The remaining VI commands were generally treated as "window dressing" by all but the most sophisticated users.

(5) There is no help facility, of any kind, provided by the VI system. At the very least, an on-line listing of commands should be provided.

APPENDIX C: EDIT

EDIT is a text editor supported by the UNIX operating system. EDIT is a simplified version of another UNIX editor and contains a minimal set of operators. It is line oriented which means that the main object of EDIT is a line of text of some finite length.

EDIT merely supports text file creation and modification operations. The user inputs text into a file by lines, indicating the end of a line by a carriage return. A display of the file will show an ordered list of lines as they exist in the file. Ordering of lines is completely determined by the system and although the user can use line numbers as a reference, the line number is not directly accessible to the user to change or set. Any display of text by EDIT is done by line. Substrings can be referenced within a line, or lines. A formatted output display by EDIT can only be achieved if the user directly inputs the desired format line by line. No processing of the contents of a line is done by EDIT.

When invoked, EDIT sets aside a temporary copy of the referenced file in a working buffer. If the file does not already exist in the directory, then it is a new file and is created. The basic set of commands available to EDIT are listed in Table C.1.

Table C.1 - EDIT Command Summary.

1. Edit a file
2. Specify a file
3. Append line(s)
4. Insert line(s)
5. Insert line(s) into an external file
6. Insert line(s) from an external file
7. Delete line(s)
8. Copy line(s)
9. Move line(s)
10. Print line(s)
11. Show line number
12. List line(s)
13. Substitute a string
14. Search for string
15. Undo last command
16. Make effect of command global
17. Move cursor
    - forward
    - backward
18. Quit

Searching for a line has the effect of making the found line the current line. Any subsequent editing operations are done in relation to the current line. Lines can be found and displayed by line numbers, and ranges of lines can be specified. Lines can also be found and displayed forward or backward, relative to the current line. A line can be found by any substring of its contents, but the entire substring must be contained in one line. Because of this deficiency a substring may not be locatable merely because it exists in the text file. When searching EDIT will move

forward or backward and will wrap around the buffer, so as to return to the starting line if the target object is not found.

New lines can be appended before the current line, or inserted after it. The user issues a command to specify that there are no more lines to add. Upon completion the current line is the last line added. Additions can also be made by moving or copying lines within the text file. Moving can be viewed as a combination of a deletion and an insertion. By specifying a range of lines to be changed, they are deleted and the system enters insert mode for the user to add the new lines. Additionally, insertions are possible from other text files.

Modifying a line is done by substituting a new string for an already existing target string on the line. If desired, the substitution can have global effect in that it will modify all occurrences of the target string on all lines.

Deletion is usually accomplished by indicating the line, or lines, to be deleted. A search command can be used with the deletion operation when the specific line numbers are not know.

EDIT protects the user from making inadvertent changes to a text file. The effects of the last executed command that effected the buffer can be reversed. Additionally, the effects of the editing session do not become permanent

unless the user issues a command to make them permanent. At that point the edited copy, which is in the buffer, replaces the original file in the directory. Leaving EDIT without indicating to make the changes permanent is like the editing session never occurred.

In addition to writing a whole buffer out to the directory, subparts can be written to another text file. This is done by specifying the range of lines and the file to be written to.

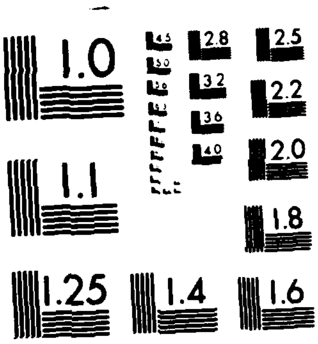
The EDIT text editor is very basic which is both an advantage and a disadvantage. It has a minimal command set and therefore is easy to learn. The biggest problem is that it is line-oriented. As such, modifications are done a line at a time, where each line is a separate entity. It does not treat the file as a whole, but as a disjoint collection of lines. It imposes the idea of line numbers, which do not exist in the text file, in order to use the editor. There are fewer high level editing operations available, as compared to current screen-oriented editors, and they are limited to operating on lines and not the text file as a whole. While capable of producing satisfactory results, due to its line at a time limits, the operation becomes tedious if the file is large, and/or there are a lot of small changes which must be done. Given the advanced features of todays line-oriented editors, EDIT is a very archaic and frustrating way to create and modify a text file.

AD-A126 979 ANALYSIS OF THE RELATIONAL DATA BASE MODEL IN SUPPORT OF AN INTEGRATED APPLICATION SOFTWARE SYSTEM(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA R NISHIMURA DEC 82

2/2

UNCLASSIFIED F/G 9/2 NL

END
DATE
FILMED
5-83
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

APPENDIX D: NROFF -ME

"NROFF -ME" is a text processing facility for files that are created on the UNIX operating system. It was created by the University of California at Berkeley, and Bell Laboratories. "NROFF" is a program that accepts an input file prepared by the user and outputs a formatted paper to the user's design. "-ME" is a macro package that enhances the capabilities of the "NROFF" program by adding additional formatting abilities and commands. The input file consists of the actual text entered by the user, through some editor system, and a series of embedded NROFF -ME commands.

There is a large vocabulary of "requests", which are really dot-commands consisting of a period followed by a two letter string. The basic NROFF package supports seventeen categories of commands, and has a total of eighty seven commands. The -ME package adds three categories and a total of sixty commands for a grand total of one hundred and forty seven commands. Table D.1 lists the NROFF and -ME command categories, and the number of commands in each.

NROFF -ME uses thirteen predefined general variables and twenty three predefined read-only variables to support its processing needs. The user is provided with a macro facility to define new commands in terms of the basic set of commands and operations on the variables. This allows the

user to abbreviate a fairly long command stream into a single command.

Table D.1 - NROFF and -ME Commands.

COMMAND CATEGORY	COMMANDS	
	NROFF	-ME
1. Font & Character Size Control	7	9
2. Page Control	7	0
3. Text Filling, Adjusting & Centering	6	0
4. Displays	0	22
5. Vertical Spacing	7	0
6. Line Length & Indenting	3	0
7. Paragraphing	0	4
8. Macros, Strings, Diversions, & Traps	13	0
9. Number Registers	3	0
10. Tabs, Leaders, & Fields	4	0
11. InputOutput Conventions	9	0
12. Hyphenation	4	0
13. Titles	3	13
14. Headings	0	6
15. Line Numbering	2	0
16. Conditional Input	8	0
17. Environment Switching	1	0
18. Standard Input Insertions	2	0
19. InputOutput File Switching	3	0
20. Miscellaneous	5	6
TOTAL	87	60

NROFF -ME is a good word processing system and it can produce some complex formatting actions. However, it does suffer from some drawbacks.

(1) Since the file is first created by the text editor and then run by NROFF, the user has a significant delay in

determining if the desired format was achieved.

(2) In addition to depending on the text editor, NROFF must depend on other programs to preprocess the text file before NROFF can handle it for specialized requests. Two examples of preprocessors are packages to handle tables and complex equation symbology. While enhancing NROFF -ME's capabilities, they add more categories and commands, and increase the amount of time necessary for the user to see the actual results of commands.

(3) The user manual for the NROFF package is not presented in sufficient detail to completely understand the effect, or use, of all commands. It appears that the user is supposed to have a basic understanding of the system before reading the manuals!

(4) The command vocabulary is fairly large and they are not easy to remember. Based on personal use, only about twenty percent of the vocabulary is generally useful and therefore remembered. Table D.2 presents a simplified listing of the most used commands.

Table D.2 - Basic Commands NROFF -ME

1. Page length
2. Line spacing
3. Line length
4. Page headers
5. Indent
   - permanent
   - temporary
6. Begin next page
7. Need # lines
8. Insert # blank lines
9. Center the next # lines
10. Break
11. Define a macro
12. Fill/No-fill
13. Hyphenate/No-hyphenate
14. Underline
15. Section/Chapter headings
16. Quotations
17. Footnotes
18. Keep an index
19. Start paragraph
   - basic
   - left adjusted
   - body indented
   - numbered
20. Start display
   - list
   - block
   - floating block
   - delayed text
21. Table handler *
   - definition
   - start
   - body
   - end
22. Equation definition
23. Multiple column format
24. Default paper formats
   - thesis
25. Control constructs
   - read special variables
   - change special register
   - conditional formatting

* part of Table preprocessor

APPENDIX E: DBASE II

DBASE II is a relational database system created by Ashton-Tate of Los Angeles, California for microcomputer systems. For this review, the CP/M version of DBASE II was used, where the DBASE II program is an executable "command file" residing in the system.

The DBASE II system utilizes several different file types: database, report form, command, index, memory, and text. Each file type has a specific purpose that is identifiable by its type name. "Report form" files store the information, specified by the user, for describing the format (headings, fields, totals, subtotals, contents, etc.) in which a "database" file is to be output. "Command" files contain a sequence of DBASE II statements, commands, and control structures necessary to create a user defined view. "Index" files are a list of pointers to a specific "database" file. "Memory" files contain the values of memory variables and constants saved previously by the user. "Text" files are collections of ASCII characters for input into a "database" file, or created by output from a "database" file. DBASE II cannot directly use "text" files. Most of the files are stored in what is known as Standard Data Format (SDF), and they can be used directly by any other program that uses SDF files. Additionally, any text

files in SDF can be used by the DBASE II system. The file is the largest data object supported by DBASE II which creates, deletes, or modifies the current file(s). A database file is brought into reference by user specification, and a maximum of two database files can be "open" at one time.

DBASE II can be used interactively or can be programmed to create a view of the database to support recurring applications. Regardless of method, DBASE II provides the user with the same basic high-level data definition (DDL) and data manipulation (DML) language. An English like command language with a very regular syntax is a user friendly feature of DBASE II. The commands are very powerful in that their operands and results are typically database files. The command structure is usually presented in the following form:

COMMAND [SCOPE] [CONDITION]

The scope modifier designates the number of records to be selected in response to the specific command. The condition modifier specifies a conditional statement that the record's field values must satisfy in order for the record to be included in the final result. Table E.1 provides a listing of the basic DBASE II commands, with duplicate commands having been factored out.

Table E.1 - DBASE II Basic Commands.

1.	Display an expression on the screen
2.	Format screen or printer output
3.	Input a character string
4.	Input a string to a memory variable
5.	Wait for user input
6.	List the records in a database
7.	Display data from a database
8.	Display the structure of a database
9.	Rename a file
10.	Erase a file
11.	Generate a report
12.	Execute a "command" file
13.	Return from a "command" file
14.	Display the contents of the memory variables
15.	Store a value in a memory variable
16.	Save memory variables to a file
17.	Restore memory variables from a file
18.	Select a specific database for use
19.	Set specific DBASE II parameters
20.	Abort a command
21.	Create a new database
22.	Edit a database
23.	Modify a database's structure, or the contents of fields in selected records
24.	Update a database from another database
25.	Add data from a text file to a database
26.	Copy data from a database to a text file
27.	Insert record(s) into a database
28.	Delete record(s) from a database
29.	Unmark records marked for deletion
30.	Locate a record based on key value, or condition
31.	Goto a specified record
32.	Move forward or backward in a database
33.	Index a database
34.	Sort a database based on a field
35.	Perform JOIN operation on two databases
36.	Count the number of records
37.	Sum a field or subfield in a database

Default ordering for records in a database file is the sequence in which the records are entered. Ordering can be altered by inserting records into specific parts of the database, and by sorting or indexing the database. In the default order, the "database" file does not contain a recognized key.

By sorting or indexing a "database" file, keys are defined and the search time required to locate a record is reduced. Multiple indexing be done for the same database, but based on different keys. Sorting produces a new "database" file, which is a copy of the original database, only it is sorted. An "indexed" file is a virtual file of pointers to the original "database" file. Whereas lookup speed can be enhanced by indexing a database, there is overhead incurred in maintenance of the "index" file. Changes made to the original database file are not reflected in the new sorted "database" or "index" file. The original database must be sorted or indexed after each change in order to remain current.

The data definition language allows the user to define the organization of the data in a new database file by specifying the name of the database, and giving information on each of its fields (name, type, width, decimal places). The structure of a new database file can also be copied from that of another database file. Additionally, new structures can be created as the result of using the JOIN operator

provided by the DBASE II system. At any time, the structure and/or contents of a file can be displayed or output. The structure of a database file can also be modified at a later time, but presents some problems in that all records currently in the database file are destroyed.

Besides using DBASE II interactively, it can be programmed in its own language through the use of "command" files. The DML statements are embedded in the file and iterative execution of DML statements are controlled by a set of DBASE II control structures (If-Then, If-Then-Else, Goto, and Do-While). "Command" files tend to make extensive use of memory variables and input/output functions which are also extensively supported by DBASE II. To create a user view the designer/programmer will edit a "command" file(s) to contain the correct DBASE II statements, commands, and control structures to manipulate the proper "database" files. The capabilities and limitations of any view is dependent on the design of the "command" file(s).

The reason for the great popularity of DBASE II is that it is a very easy database management system to learn and use. Its English-like command language is natural and user friendly. Although the command set is rather extensive, the command names accurately describe their action and use a regular syntax so they are easy to remember. The interactive nature and full screen display orientation makes user interaction simple and direct. With its set of

predefined functions, input/output commands, "command" files, and programming constructs it is easy to create views for almost any application. DBASE II is a powerful relational database system yet it is obvious that the designers gave much thought to keeping it simple and did not introduce complexity for its own sake. However, there are a couple of problems witn DBASE II which are worth mentioning, and they are all probaoly due to the justified emphasis on simplicity.

(1) At any one time, a maximum of two databases can be in reference. This limitation requires that databases be explicitly brought into and out of use. It would help if there was another method, besides using a "command" file, for performing operations on multiple taoles.

(2) In modifying the structure of a database the contents are deleted. This requires that the database be explicitly saved to an external database and then be recopied back after structure modification. It is an inconvenience, to say the least.

(3) The only relational operation directly provided by the system is the JOIN command. It would greatly enhance the capability of the system to provide more of the operators.

(4) The display structure is a little bit too rigid, and the user does not have much direct control, sort of writing a "command" file, to effect the output format.

APPENDIX F: SEQUITUR

SEQUITUR is a relational database system designed by the Pacific Software Manufacturing Company of Berkeley, California.

SEQUITUR sees a database as a collection of named tables, each of which contains some kind of data related to the subject of the database. Each database has a set of system tables. The "Column" table lists the name, type, size, and display format of all columns authorized for use in the database's tables. The "Table" table lists the names of the columns that are included in each of the database's tables. Together the "Column" and "Table" tables act as part of a data dictionary system for the database.

SEQUITUR has a fairly large command vocabulary of over sixty seven commands. There are twenty five basic commands, forty two screen editor commands, and more formed by combinations of the previous commands. A multilevel "Help" facility is used to support the user.

SEQUITUR offers four kinds of help. There are status lines at the top of the screen. An "edit card" display can be called by the user in order to see a comprehensive list of cursor object and motion keys, and escape operations. The "help" command summons an on-line manual, that is preset by the user to provide no, medium, or maximum help. Lastly,

there are situational help prompts that occur during the command process.

Table F.1 - SEQUITUR Basic Commands.

1.	CHOOSE (database)
2.	CREATE (database)
3.	ADD to (table)
4.	EDIT (table)
5.	SHOW (table)
6.	PRINT (table)
7.	REPORT generator
8.	FORMS generator
9.	SELECT from (table) *
10.	MANUAL select
11.	JOIN (tables)
12.	SORT (tables) *
13.	UNION *
14.	INTERSECTION *
15.	DIFFERENCE *
16.	UNIQUE rows *
17.	DUPLICATE rows *
18.	COPY
19.	APPEND
20.	REMOVE rows
21.	RENAME column
22.	COMPACT base
23.	DUMP to (file)
24.	LOAD from (file)
25.	HELP from manual
26.	EXIT

* = Member of SEQUITUR's "set" commands.

The twenty five basic commands cover the major operational capabilities of the SEQUITUR system. The commands are presented to the user in the form of a menu,

there are situational help prompts that occur during the command process.

Table F.1 - SEQUITUR Basic Commands.

```
------------------------
1.  CHOOSE (database)
2.  CREATE (database)
3.  ADD to (table)
4.  EDIT (table)
5.  SHOW (table)
6.  PRINT (table)
7.  REPORT generator
8.  FORMS generator
9.  SELECT from (table) *
10. MANUAL select
11. JOIN (tables)
12. SORT (tables) *
13. UNION *
14. INTERSECTION *
------------------------
15. DIFFERENCE *
16. UNIQUE rows *
17. DUPLICATE rows *
18. COPY
19. APPEND
20. REMOVE rows
21. RENAME column
------------------------
22. COMPACT base
23. DUMP to (file)
24. LOAD from (file)
25. HELP from manual
26. EXIT
------------------------
```

* = Member of SEQUITUR's "set" commands.

The twenty five basic commands cover the major operational capabilities of the SEQUITUR system. The commands are presented to the user in the form of a menu,

and once a choice is made SEQUITUR enters the display mode necessary to support that choice. Table F.1 lists the basic commands, plus the command for exiting from SEQUITUR.

The SEQUITUR display modes are organized as "tables", or "pages". The table mode is similar to the approach taken by the "Query-by-Example" system (QBE), and presents the data in columns and rows with vertical lines separating the columns and indicators for new rows. Alternatively, the page mode presents the data one row at a time, with the column headings listed vertically. The user has the ability to flip back and forth between the two display modes at will.

Table F.2 - SEQUITUR Cursor Object & Motion Commands.

1. Move cursor up one line
2. Move cursor down one line
3. Move cursor left one object
4. Move cursor to next object
5. Move cursor to beginning of object
6. Move cursor to previous word
7. Move cursor to end of current object
8. Move cursor to next word
9. Object = word
10. Object = line
11. Object = sentence
12. Object = paragraph
13. Object = view
14. Object = page or screen
15. Object = column
16. Object = row
17. Object = one character

Once in a desired display mode the user must make use of the editor commands to make changes to the table. All editor commands are single keys combined with the <Control>, <Escape>, or <Tab> keys. Table F.2 provides a list of the cursor object and motion commands available. Most operations require two commands since the object must be specified first, and then the actual operation.

Table F.3 - SEQUITUR Screen Editor Commands.

1. Delete left portion of object
2. Delete entire object
3. Delete right portion of object
4. Flips "insert" toggle
5. Shows rows marked for deletion
6. Flip "page-table" display style
7. Goto #-th object
8. Goto last object
9. Restores more recent version of row
10. Display earlier version of row
11. Executes a command
12. Search forward for column entry
13. Search backwards for column entry
14. Edit card display

The screen editor commands are used to make actual changes (additions, modifications, or deletions) to the displayed table on the screen. Table F.3 lists these commands which are used in conjunction with the cursor object and movement commands listed previously.

Additionally there are a number of miscellaneous commands that are provided to aid the user. These are listed in Table F.4.

Table F.4 - Additional SEQUITUR Commands

1. Get Edit Help
2. Scroll Forward
3. Scroll Backwards
4. Interrupt Present Operation
5. Lock/Unlock Cursor Object

There are an abundance of table types in SEQUITUR. "Virtual" tables consist of pointers to data in a "base" table(s), and are formed by conducting relational operations (e.g. JOIN) on the base table(s). Virtual tables are permanent additions to the database. All operations conducted on the virtual table effect the base table, but not all operations on the base table will reflected in the virtual table.

"Slice" tables consist of the data from a "home" table, and are formed by restricting or rearranging the columns in the home table. Actually, slice tables are just alternate ways of viewing the same home table. All operations conducted on the slice table effect the home table, and all operations on the home table effect the slice table.

"Template" tables are used to store control information on the operation(s) (SELECT, SORT, UNION, DUPLICATE, UNIQUE, INTERSECTION, and DIFFERENCE) desired to be performed on a set of "base" tables. The user specifies once the sequence of operations to be performed, and each time that result is desired the appropriate template table is called to create the desired virtual table.

SEQUITUR provides several methods of outputting data to the user:

(1) There is the "print" command which prompts the user to specify heading, page length, margins, page number, date, column/row divider symbol, etc. for either a "table" or "page" style output. The entire table is then output, one record at a time, in the specified format.

(2) There is the "form generator". The user creates a form letter or document by making an entry in the "forms" table in either "page" or "table" style, and answering several system prompts as to page size, width, margins. The form generator is intended for letter type generation since it only allows one text field in the form. All other entries are pulled from an appropriate table and the "form" repeated for each row in that table.

(3) There is the "report generator". The user creates a report table that is associated with a known data table. The report table specifies which data table columns are to be used, how they are positioned, what name they have on the

form, allotted width, and alignment. Again, the user must specify formatting items like page length, line length, margins, delimiters, and other related items. The individual columns in the report table can be marked for sorting, grouping, and/or arithmetic processing. If arithmetic processing is opted for, then another table, the "function" table is created to record what is to be done to each column - total, minimum, maximum, average, or count.

Based on a very short familiarization experience with SEQUITUR there is no doubt that it is a powerful and complete relational DBMS. However, it is not as user friendly as its advertisements would lead you to believe. Some of the problems encountered were:

(1) Too many commands to remember. This increased learning time and added to the confusion. Too many of the commands were just window dressing in that their effect could have be done using other commands. (Like the "Object =", extra cursor movement and deletion commands.) While using keys as commands leads to faster command input, it makes things more difficult when there are so many commands the symbol on the key has little or no relation to its effect.

(2) The structure of the user interface was unwieldy. It was easy to get lost and difficult to recover to a known location. Operations that worked under one condition did

not work in another, or produced completely different and unexpected results. (e.g. in some instances the "execute" command will return you to the main menu, in others it was ignored or treated as a mistake.)

(3) There were too many types of tables, ways of using tables, editing tables, and creating relations between tables. The user is being swamped with a level of detail that is better left to the system. It seems that SEQUITUR was created with simplicity and user support being lesser considerations to system sophistication.

APPENDIX G: VISICALC

VISICALC is an electronic spreadsheet program created by Software Arts, Inc. of Cambridge, Massachusetts and marketed by Personal Software Inc. of Sunnyvale, CA. Its purpose is to allow the user to easily model a wide range of numerical problems in a standard tabular format by replacing the user's pencil, calculator, and scratchpad.

The screen is divided into a grid of columns and rows that form addressable (column, row) entry positions. The columns, which run across the top of the grid, are lettered starting with "A" and the rows, which run down the side, are numbered starting with "1". Each entry position is an independent entity, and can contain a character string, a numeric value, or a function that must be calculated. Entry positions that contain functions are recalculated by VISICALC each time certain conditions are met. The functions will specify values in terms of constants, operators, and the values of other entry positions.

The screen is used as a "window" into the spreadsheet and is modifiable by the user. The user is given numerous commands, see Table G.1, with which to alter the display format of the screen.

Table G.1 - VISICALC Display Commands.

1. Clear Spread Sheet
2. Set Global Display Format To;
   - Integer
   - Dollars & Cents
   - Left/Right Justified
   - Graph
3. Set Entry Display Format To;
   - Integer
   - Dollars & Cents
   - Left/Right Justified
   - Graph
4. Reset Entry To Global Display Format
5. Set Column Width Within A Window
6. Set Order Of Recalculation;
   - Column Wise
   - Row Wise
7. Set Recalculation;
   - Automatic
   - Manual
8. Move An Entire Row Or Column
9. Window Control;
   - Split Screen Horizontal
   - Split Screen Vertical
   - Single Window
10. Window Synchronization;
    - Synchronized
    - Unsynchronized

The window can be "split" into two halves so as to look into nonadjoining areas of the spread-sheet simultaneously. The two windows can be "synchronized" so they move together, or unsynchronized so movement is independent. Display format may be globally set for the screen as a whole, or individual entry positions can be assigned their own format. Column width is variable from 3 to 37, but columns in the

same window must have the same width. The value of each entry position is calculated by "column order" (A1, A2, ..., An, B1, B2, ..., Bn, C1, etc.) unless the user changes the recalculation order to "row order" (A1, B1, ..., n1, A2, B2, ..., n2, C2, etc.). By default VISICALC starts in "automatic" recalculation mode where the value of all entry positions are recalculated each time an entry is changed. As this can significantly slow down the model when large grids and/or complicated numerical expressions are used, the user can enter "manual" recalculation mode where a command must be issued to cause recalculation to occur.

VISICALC provides a command-line oriented editor that enters, modifies, or deletes data in a referenced entry position(s). A cursor is provided on the grid to indicate the current entry position referenced by VISICALC. There are screen commands to allow the user to scroll across the grid or to move to an exact (row, column) entry position. If needed, the numeric processing capability of VISICALC can be used like a calculator to support the user's computational needs. A powerful capability of VISICALC is the replicate command. This allows the user to define an entry once, and then have it entered in a range of successive column or row entry positions. Additionally, the user can specify if the original entry is to be replicated exactly, or should any references to other entry positions

be updated at each new position to take into account relative position on the spreadsheet.

Table G.2 - VISICALC Cursor Movement & Entry Commands.

11. Move Cursor Right Or Up
12. Move Cursor Left Or Down
13. Change Cursor Direction;
    - Up/Down
    - Right/Left
14. Move Cursor To The Other Window
15. Move Cursor To A Specific Entry Position

---

16. Abort Last Command
17. Set An Entry Position To Blank
18. Delete An Entire Row Or Column
19. Inset A New Row Or Column
20. Replicate An Entry
21. Set Title Areas;
    - Horizontal Title
    - Vertical Title
    - No Title
22. Repeat A Label Entry
23. Make An Immediate Numerical Calculation
24. Enter A Label In An Entry Position
25. Enter A Value In An Entry Position
26. Save A Copy Of The Spread-Sheet

Since VISICALC is a numerical modeling tool it has a series of arithmetic and aggregate functions that it supports. Table G.3 provides a listing. VISICALC has been designed to store numbers in decimal format, not binary, and maintains them with up to eleven significant digits or decimal places.

Table G.3 - VISICALC Arithmetic & Aggregate Functions

a. Addition
b. Subtraction
c. Multiplication
d. Division
e. Exponentiation
f. Calculate The Sum Of A Range Of Values
g. Calculate The Minimum In A Range Of Values
h. Calculate The Maximum In A Range Of Values
i. Count The Number Of Entries In A List
j. Calculate The Average Of A Range Of Values
k. Calculate The Net-Present-Value Of A Range Of Values
l. Perform A Lookup Operation
m. PI (3.1415926536)
n. Calculate The Absolute Value
o. Calculate The Integer Portion Of A Value
p. Square Root
q. Logarithms, Base 2
r. Logarithms, Base 10
s. Trigonometric Functions (Sin, Cos, Tan, Asin, Acos, Atan)

VISICALC makes use of dynamic memory allocation so the actual dimensions of the spread-sheet depend on the amount of memory available and the complexity of the entries made by the user. The user does not have to worry about memory allocation since VISICALC takes responsibility for its use and efficiency. As entries shrink, or are deleted, VISICALC reclaims the extra memory space. The user is shown how much memory remains and a warning prompt occurs when memory space is nearly exhausted.

For a permanent copy of the contents of the spread sheet the user may send the output to a printer. A subpart of the total spread-sheet may be sent by designating the lower right corner to be printed.

VISICALC is a powerful and fairly simple modeling tool whose advantages seem to easily outweigh the disadvantages. The command vocabulary is low (26 commands, 19 functions) and the greater majority are actually useful and not just window dressing. The user manual is well written and easily understood, but is fairly long. VISICALC supports a known human weakness (small/fast short term memory, large/slow long term memory, and slow calculation speed) by remembering the details of a commonly reoccurring user problem (the situation to be modeled), limiting the user to providing a smaller and more select set of initial inputs, and performing the computations in a faster, more reliable, and repeatable manner. However it does have some problems:

(1) Command strings and their effect must be memorized since there is little relation to the string and the effect. Menus provided by the system are very poor, and require you to already know the meaning of the command string.

(2) A basic understanding of VISICALC and a high degree of operational capability can be obtained, in a fairly short time, by reading only the first third of the user manual. However, to gain maximum use of the system requires a

significant amount of time and effort to read the entire user manual and experiment with the operations. Some nice to know features that have a major effect on model validity (e.g. recalculation order) are discussed at the end of the user manual and might be easily missed.

APPENDIX H: ZIP

The relational data base management system "DBASE II", described in Appendix (D), contains a set of commands which, when embedded in a "command" file, define the output format used to generate the display on the screen, or output to the printer. In addition to generating the display form, the commands also direct the DBASE II system to either determine the values of the entries from a record in the referenced database, or from memory variables. If the input device is the screen/keyboard, DBASE II may retrieve a user entered value from the screen and store it in a field of a database record, or in a memory variable. These form definition commands can also be put into a new type of file, the "format" file, by ZIP. In this case the format, contained in the "format" file, is used as an display overlay to prompt the user to change data values in an existing record in a "database" file.

ZIP is a CP/M program used to generate, or modify, a DBASE II "command" or "format" file. It is a powerful tool in the sense that the user is not required to know the details of the DBASE II form generation capability ("command" files, and display commands). ZIP presents the user with a blank screen and an on-screen editor, which supports several levels of cursor movement and formatting

commands, to help in the form design. Table H.1 lists the ZIP editor commands.

Table H.1 - ZIP Editor Commands.

1. Screen commands
   - top
   - bottom
   - next
   - previous
   - first
   - last
2. Middle of line
3. Insert a space
4. Add a line
5. Delete
   - character
   - line
6. Draw/Erase horizontal line
7. Draw/Erase vertical line
8. Erase/Save work file
9. Insert DBASE II command expression
10. Change variable
    - vertical marker
    - horizontal marker
    - tab spacing
    - margin
    - page length
11. Quit

The cursor can be moved to any position on the blank screen where the user will enter the information required by the ZIP program. Information is conveniently limited to literal strings, memory variables, record field values, and fetching a value from the screen and storing it into a record field or memory variable. Interspersed between these ZIP

formatting commands may be DBASE II executable commands if the file type is "command". There are special purpose commands to draw, or undraw, vertical and horizontal lines on the form.

The ZIP program may be viewed as a translator between the screen design made by the user and the operations of DBASE II. The screen contents associated with each screen position are translated into a sequence of DBASE II commands, statements, and control structures which are organized as either a "command" or "format" file. ZIP also places any embedded execution commands into the file and automatically sets, or resets, the appropriate system "toggles" as needed.

ZIP is a useful support tool for DBASE II in that it relieves the user from having to program a "command" file in order to create a desired display format. However, it must be pointed out that ZIP is a very basic formatter, is line oriented, and is incapable of the more complex types of displays.

APPENDIX I: MAIL

"MAIL" is an electronic mail facility produced by the University of California at Berkeley and Bell Laboratories for the UNIX operating system. It allows users to send messages to other users, or groups of users, on the system.

The basic unit of the MAIL system is the message, which is simply a special type of text file. The message is preformatted and contains fields for originator, destination, subject, copy to, and body. Messages are contained either in the users "private" mailbox or in the "system" mailbox. A "dead-letter" file is also maintained for each user to contain messages which cannot be delivered to a valid destination. The private mailbox and dead-letter file are maintained as text files in the UNIX directory and therefore can be used by other programs running under UNIX.

Upon logging into the UNIX system, a prompt appears at the terminal indicating that there is mail for the user. Messages addressed to a user are initially contained in the system mailbox, and can be read from the system mailbox by the MAIL facility. The messages already in the private mailbox and/or dead-letter file are text files and thus not directly accessible to the MAIL facility.

The user may elect to read the mail by invoking the MAIL facility. A one line summary of all messages in the system

mailbox is presented to the user, and each message is given an integer identification number starting at one. At this point the user has a number of different options available as summarized in Table I.1.

Table I.1 - MAIL Command Summary

1. Alias a name +
2. Unalias a name(s)
3. Goto previous message * +
4. Goto next message * +
5. Display summary of commands +
6. Display out all currently defined aliases
7. Display a message
8. Display out headers of message list +
9. Display message list
10. Display size of each message
11. Display top few lines of each message
12. Execute the following UNIX shell command
13. Change directory
14. Delete message(s) +
15. Delete current message, print next message
16. Undelete messages marked for deletion
17. Reply to a received message *
18. Edit a list of messages in turn
19. Send message to designated users +
20. End-of-message +
21. Exit, don't change system mailbox *
22. Quit, save undeleted or unsaved messages in the user's mailbox, save unreferenced in the system mailbox.
23. Mark message(s) to be saved in system mailbox *
24. Save a message list by appending to a text file *
25. List current range of message headers
26. Help +
27. Set options +
28. Unset options

* MAIL facility has more than one command to perform this action.

The user may select a message and read it. After reviewing the message the user may forget the message, save it in the system mailbox, delete it, or prepare a response. When the user quits the MAIL facility all messages which have not been deleted, saved, or reviewed are placed back into the system mailbox. The remaining messages, those reviewed but no special action indicated, are placed in the private mailbox. If the user desires, the MAIL facility can be exited and the system mailbox left unchanged. Additionally the user can create "alias" names that correspond to multiple users, ask for message summaries, append messages to files, or invoke an editor.

The MAIL utility does not contain its own editor, but depends on the editor(s) available to the UNIX system and on the user to set an option specifying which one is desired. When the user indicates that a message is to be created, the editor is invoked, the user enters the text, and when finished issues an end-of-message command to return control to the MAIL facility. While in the editor, the user can issue "escape" commands that directly effect the message processing. A listing of these escape commands is provided in Table I.2. Contents of other files may be inserted into the message, names of recipients added or changed, the header field edited, or an alternate editor invoked.

Table I.2 - MAIL escape commands

1.	Execute UNIX shell command
2.	Add names to recipients of copy
3.	Read "deadletter" file into message
4.	Invoke text editor
5.	Abort the message being sent
6.	Insert a named file into the message +
7.	Create a subject field
8.	Write the message into a named file
9.	Pipe the message through a process as a filter
10.	Insert a string into the message

While in the MAIL facility, UNIX shell commands may be issued. The MAIL facility is temporarily interrupted, the command is executed, and then the MAIL facility is resumed without adverse effect.

Table I.3 - MAIL options.

1.	(Append/Prepended) messages to private mailbox
2.	(Yes/No) Subject line prompt
3.	(Yes/No) Prompt for carbon copy recipients of message
4.	(Yes/No) Modify delete command
5.	(Yes/No) Ignore terminal interrupt signals
6.	(Yes/No) Include sender in group message recipients
7.	(Yes/No) Saving interrupted messages
8.	Define default editor name
9.	Define escape character
10.	Define file to record outgoing mail
11.	Define number of lines in the "top" of a message

Additionally, the MAIL facility has a series of options the user can change to tailor its operation. Table I.3 provides a listing of these options.

The MAIL facility is a good support program and is quite capable of accomplishing its goals. However, it has more than its fair share of problems.

(1) There is a very limited user manual, and experience must be gained from other users or by trial and error.

(2) There are too many commands, and too many of those duplicate each other. The number of commonly useful commands is low (marked with a +), with the rest being window-dressing.

(3) The facility is not user friendly. The user must be aware of location in the facility and what is expected next, because there are no special prompts and the help command only provides a command summary.

(4) If the message recipient is on line when the message arrives, whatever operation is in progress is rudely interrupted by the display of the message. This can be very disconcerting to the recipient.

(5) The user can't determine which message is going where (system mailbox, private mailbox, dead-letter file), prior to leaving the MAIL facility.

BIBLIOGRAPHY

Bricklin, D. & Franklin, B., VISICALC User Manual, Personal Software, Inc. 1979

Codd, E., " Relational Database: A Practical Foundation for Productivity", Communications of the ACM, 25, 2, pp. 109 - 117, (Feb 1982)

DBASE II User Manual, Ashton Tate 1981

Ghosh, S., Data Base Organization for Data Management, Academic Press 1977

Horowitz, E. & Sahni, S., Fundamentals of Data Structures, Computer Science Press, Inc. 1976

Kent, W., Data and Reality, North Holland Publishing Co. 1978

Naiman, A., Introduction to Word Star, Sybex Inc. 1982

SEQUITUR User Manual, Pacific Software Manufacturing Company 1982

UNIX Programmer's Manual, Seventh Edition, Volume 2A Bell Telephone Labratories, Inc 1979

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Professor Dusan Z. Badal, Code 52ZD Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
4.	LT Rodney Nishimura, USN 14541 S. Catalina Ave. Gardena, California 90247	1

DATE

ILME